The Need for Data Structures

Data structures organize data ⇒ more efficient programs.
More powerful computers ⇒ more complex applications.
More complex applications demand more calculations.
Complex computing tasks are unlike our everyday experience.

Organizing Data

Any organization for a collection of records can be searched, processed in any order, or modified.
The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be efficient if it solves the problem within its resource constraints.
- Space
- Time

- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:
1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some Questions to Ask

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?
Data Structure Philosophy

Each data structure has costs and benefits. Rarely is one data structure better than another in all situations.

A data structure requires:
- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Data Structure Philosophy (cont)

Each problem has constraints on available space and time. Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:
- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

Goals of this Course

1. Reinforce the concept that costs and benefits exist for every data structure.
2. Learn the commonly used data structures.
   - These form a programmer's basic data structure "toolkit."
3. Understand how to measure the cost of a data structure or program.
   - These techniques also allow you to judge the merits of new data structures that you or others might invent.

Abstract Data Types

Abstract Data Type (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

Each ADT operation is defined by its inputs and outputs.

Encapsulation: Hide implementation details.

Data Structure

- A data structure is the physical implementation of an ADT.
  - Each operation associated with the ADT is implemented by one or more subroutines in the implementation.
- Data structure usually refers to an organization for data in main memory.
- File structure is an organization for data on peripheral storage, such as a disk drive.

Metaphors

An ADT manages complexity through abstraction: metaphor.
- Hierarchies of labels
Ex: transistors ⇒ gates ⇒ CPU.

In a program, implement an ADT, then think only about the ADT, not its implementation.
Logical vs. Physical Form

Data items have both a logical and a physical form.

Logical form: definition of the data item within an ADT.
- Ex: Integers in mathematical sense: +, -

Physical form: implementation of the data item within a data structure.
- Ex: 16/32 bit integers, overflow.

Problems

- Problem: a task to be performed.
  - Best thought of as inputs and matching outputs.
  - Problem definition should include constraints on the resources that may be consumed by any acceptable solution.

Problems (cont)

- Problems ⇔ mathematical functions
  - A function is a matching between inputs (the domain) and outputs (the range).
  - An input to a function may be single number, or a collection of information.
  - The values making up an input are called the parameters of the function.
  - A particular input must always result in the same output every time the function is computed.

Algorithms and Programs

Algorithm: a method or a process followed to solve a problem.
- A recipe.

An algorithm takes the input to a problem (function) and transforms it to the output.
- A mapping of input to output.

A problem can have many algorithms.

Algorithm Properties

An algorithm possesses the following properties:
- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.

A computer program is an instance, or concrete representation, for an algorithm in some programming language.
**Mathematical Background**

Set concepts and notation.
Recursion
Induction Proofs
Logarithms
Summations
Recurrence Relations

**Estimation Techniques**

Known as “back of the envelope” or “back of the napkin” calculation

1. Determine the major parameters that effect the problem.
2. Derive an equation that relates the parameters to the problem.
3. Select values for the parameters, and apply the equation to yield and estimated solution.

**Estimation Example**

How many library bookcases does it take to store books totaling one million pages?

Estimate:
- Pages/inch
- Feet/shelf
- Shelves/bookcase

**Algorithm Efficiency**

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer’s resources.

**Algorithm Efficiency (cont)**

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm’s cost?

**How to Measure Efficiency?**

1. Empirical comparison (run programs)
2. Asymptotic Algorithm Analysis

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on “size” of the input.

Running time is expressed as \( T(n) \) for some function \( T \) on input size \( n \).
Examples of Growth Rate

Example 1.

```c
// Find largest value
int largest(int array[], int n) {
    int currlarge = 0; // Largest value seen
    for (int i=1; i<n; i++) // For each val
        if (array[currlarge] < array[i])
            currlarge = i;      // Remember pos
    return currlarge;       // Return largest
}
```

Examples (cont)

Example 2: Assignment statement.

Example 3:

```c
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<n; j++)
        sum++;
```

Growth Rate Graph

Best, Worst, Average Cases

Not all inputs of a given size take the same
time to run.

Sequential search for $K$ in an array of $n$
integers:

- Begin at first element in array and look at
each element in turn until $K$ is found

Best case:
Worst case:
Average case:

Which Analysis to Use?

While average time appears to be the fairest
measure, it may be difficult to determine.

When is the worst case time important?

Faster Computer or Algorithm?

What happens when we buy a computer 10
times faster?

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>$n$</th>
<th>$n'$</th>
<th>Change</th>
<th>$n'/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10n$</td>
<td>1,000</td>
<td>10,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$20n$</td>
<td>500</td>
<td>5,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$5n \log n$</td>
<td>250</td>
<td>1,842</td>
<td>$\sqrt{10} n &lt; n' &lt; 10n$</td>
<td>7.37</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>70</td>
<td>223</td>
<td>$n' = \sqrt{10n}$</td>
<td>3.16</td>
</tr>
<tr>
<td>$2^n$</td>
<td>13</td>
<td>16</td>
<td>$n' = n + 3$</td>
<td>-----</td>
</tr>
</tbody>
</table>
Asymptotic Analysis: Big-oh

Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $O(f(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

Usage: The algorithm is in $O(n^2)$ in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps in [best, average, worst] case.

Big-oh Notation (cont)

Big-oh notation indicates an upper bound.

Example: If $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.

Wish tightest upper bound:
While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

Big-Oh Examples

Example 1: Finding value $X$ in an array (average cost).

$T(n) = c_x n / 2$.
For all values of $n > 1$, $c_x n / 2 \leq c_x n$.
Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_x$.

Big-Oh Examples

Example 2: $T(n) = c_1 n^2 + c_2 n$ in average case.

$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$.

Therefore, $T(n) \leq cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.
Therefore, $T(n)$ is in $O(n^2)$ by the definition.

Example 3: $T(n) = c$. We say this is in $O(1)$.

A Common Misunderstanding

"The best case for my algorithm is $n=1$ because that is the fastest." WRONG!

Big-oh refers to a growth rate as $n$ grows to $\infty$.

Best case is defined as which input of size $n$ is cheapest among all inputs of size $n$.

Big-Omega

Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \geq cg(n)$ for all $n > n_0$.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $cg(n)$ steps.

Lower bound.
### Big-Omega Example

\[ T(n) = c_1 n^2 + c_2 n. \]

For all \( n > 1 \),
\[ c_1 n^2 + c_2 n = c_1 n^2. \]
\[ T(n) \geq c_1 n^2 \] for \( c = c_1 \) and \( n_0 = 1 \).

Therefore, \( T(n) \) is in \( \Omega(n^2) \) by the definition.

We want the greatest lower bound.

### Theta Notation

When big-O and \( \Omega \) meet, we indicate this by using \( \Theta \) (big-Theta) notation.

**Definition:** An algorithm is said to be \( \Theta(h(n)) \) if it is in \( O(h(n)) \) and it is in \( \Omega(h(n)) \).

### A Common Misunderstanding

Confusing worst case with upper bound.

Upper bound refers to a growth rate.

Worst case refers to the worst input from among the choices for possible inputs of a given size.

### Simplifying Rules

1. If \( f(n) \) is in \( O(g(n)) \) and \( g(n) \) is in \( O(h(n)) \), then \( f(n) \) is in \( O(h(n)) \).
2. If \( f(n) \) is in \( O(kg(n)) \) for any constant \( k > 0 \), then \( f(n) \) is in \( O(g(n)) \).
3. If \( f_1(n) \) is in \( O(g_1(n)) \) and \( f_2(n) \) is in \( O(g_2(n)) \), then \( f_1(n) + f_2(n) \) is in \( O(max(g_1(n), g_2(n))) \).
4. If \( f_1(n) \) is in \( O(g_1(n)) \) and \( f_2(n) \) is in \( O(g_2(n)) \) then \( f_1(n)f_2(n) \) is in \( O(g_1(n)g_2(n)) \).

### Running Time Examples (1)

**Example 1:**
\[ a = b; \]
This assignment takes constant time, so it is \( \Theta(1) \).

**Example 2:**
\[ \text{sum} = 0; \]
\[ \text{for} \ (i=1; \ i<=n; \ i++) \]
\[ \text{sum} += n; \]

### Running Time Examples (2)

**Example 3:**
\[ \text{sum} = 0; \]
\[ \text{for} \ (i=1; \ i<=n; \ i++) \]
\[ \text{for} \ (j=1; \ j<=i; \ j++) \]
\[ \text{sum}++; \]
\[ \text{for} \ (k=0; \ k<n; \ k++) \]
\[ A[k] = k; \]
Running Time Examples (3)

Example 4:

```c
sum1 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum1++;
sum2 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    sum2++;
```

Running Time Examples (4)

Example 5:

```c
sum1 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=n; j++)
    sum1++;
sum2 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=k; j++)
    sum2++;  
```

Binary Search

// Return position of element in sorted
// array of size n with value K.
int binary(int array[], int n, int K) {
  int l = -1;  
  int r = n;  // l, r are beyond array bounds
  while (l+1 != r) {  // Stop when l, r meet
    int i = (l+r)/2;  // Check middle
    if (K < array[i]) r = i;  // Left half
    if (K == array[i]) return i;  // Found it
    if (K > array[i]) l = i;  // Right half
  }
  return n;  // Search value not in array
}

Other Control Statements

`while` loop: Analyze like a `for` loop.

`if` statement: Take greater complexity of `then/else` clauses.

`switch` statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.

Analyzing Problems

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for every possible algorithm.
Analyzing Problems: Example

Common misunderstanding: No distinction between upper/lower bound when you know the exact running time.

Example of imperfect knowledge: Sorting
1. Cost of I/O: $\Omega(n)$.
2. Bubble or insertion sort: $O(n^2)$.
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$.
4. We prove later that sorting is $\Omega(n \log n)$.

Multiple Parameters

Compute the rank ordering for all C pixel values in a picture of P pixels.

```
for (i=0; i<C; i++)  // Initialize count
    count[i] = 0;
for (i=0; i<P; i++)  // Look at all pixels
    count[value(i)]++; // Increment count
sort(count);         // Sort pixel counts
```

If we use $P$ as the measure, then time is $\Theta(P \log P)$.
More accurate is $\Theta(P + C \log C)$.

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space Data Structure

Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.
- Encoding or packing information
  - Boolean flags
- Table lookup
  - Factorials

Disk-based Space/Time Tradeoff Principle:
The smaller you make the disk storage requirements, the faster your program will run.

Lists

A list is a finite, ordered sequence of data items.

Important concept: List elements have a position.

Notation: $<a_0, a_1, \ldots, a_{n-1}>$
What operations should we implement?

List Implementation Concepts

Our list implementation will support the concept of a current position.

We will do this by defining the list in terms of left and right partitions.
- Either or both partitions may be empty.

Partitions are separated by the fence.

$<20, 23 \mid 12, 15>$
List ADT

template <class Elem> class List {
public:
    virtual void clear() = 0;
    virtual bool insert(const Elem&) = 0;
    virtual bool append(const Elem&) = 0;
    virtual bool remove(Elem&) = 0;
    virtual void setStart() = 0;
    virtual void setEnd() = 0;
    virtual void prev() = 0;
    virtual void next() = 0;
    virtual int leftLength() const = 0;
    virtual int rightLength() const = 0;
    virtual bool setPos(int pos) = 0;
    virtual bool getValue(Elem&) const = 0;
    virtual void print() const = 0;
};

List ADT Examples

List: <12 | 32, 15>
MyList.insert(99);
Result: <12 | 99, 32, 15>
Iterate through the whole list:
for (MyList.setStart(); MyList.getValue(it); MyList.next())
    DoSomething(it);

List Find Function

// Return true iff K is in list
bool find(List<int>& L, int K) {
    int it;
    for (L.setStart(); L.getValue(it); L.next())
        if (K == it) return true;  // Found it
    return false;                // Not found
}

Array-Based List Insert

Array-Based List Class (1)

template <class Elem> // Array-based list class
AList : public List<Elem> {
private:
    int maxSize;    // Maximum size of list
    int listSize;   // Actual element count
    int fence;      // Position of fence
    Elem* listArray; // Array holding list
public:
    AList(int size=DefaultListSize) {
        maxSize = size;
        listSize = fence = 0;
        listArray = new Elem[maxSize];
    }
}
Array-Based List Class (2)

```cpp
~AList() { delete[] listArray; }
void clear() {
    delete[] listArray;
    listSize = fence = 0;
    listArray = new Elem[maxSize];
}
void setStart() { fence = 0; }
void setEnd() { fence = listSize; }
void prev() { if (fence != 0) fence--; }
void next() { if (fence <= listSize) fence++; }
int leftLength() const { return fence; }
int rightLength() const { return listSize - fence; }
```

Array-Based List Class (3)

```cpp
bool setPos(int pos) {
    if ((pos >= 0) && (pos <= listSize))
        fence = pos;
    return (pos >= 0) && (pos <= listSize);
}
bool getValue(Elem& it) const {
    if (rightLength() == 0) return false;
    else {
        it = listArray[fence];
        return true;
    }
}
```

Insert

// Insert at front of right partition
template<class Elem>
bool AList<Elem>::insert(const Elem& item) {
    if (listSize == maxSize) return false;
    for(int i=listSize; i>fence; i--)
        // Shift Elems up to make room
        listArray[i] = listArray[i-1];
    listArray[fence] = item;
    listSize++; // Increment list size
    return true;
}

Append

// Append Elem to end of the list
template<class Elem>
bool AList<Elem>::append(const Elem& item) {
    if (listSize == maxSize) return false;
    listArray[listSize++] = item;
    return true;
}

Remove

// Remove and return first Elem in right
// partition
template<class Elem>
bool AList<Elem>::remove(Elem& it) {
    if (rightLength() == 0) return false;
    it = listArray[fence]; // Copy Elem
    for(int i=fence; i<listSize-1; i++)
        // Shift them down
        listArray[i] = listArray[i+1];
    listSize--; // Decrement size
    return true;
}

Link Class

Dynamic allocation of new list elements.

// Singly-linked list node
template<class Elem>
class Link {
public:
    Elem element; // Value for this node
    Link* next; // Pointer to next node
    Link(const Elem& elemval, // Link(const Elem& elemval, 
        Link* nextval =NULL) // Link* nextval =NULL)
    { element = elemval; next = nextval; }
    Link(Link* nextval =NULL) // Link(Link* nextval =NULL)
    { next = nextval; }
};
// Linked list implementation

template <class Elem> class LList:
public List<Elem> {
private:
    Link<Elem>* head; // Point to list header
    Link<Elem>* tail; // Pointer to last Elem
    Link<Elem>* fence; // Last element on left
    int leftcnt; // Size of left
    int rightcnt; // Size of right

    void init() { // Initialization routine
        fence = tail = head = new Link<Elem>;
        leftcnt = rightcnt = 0;
    }

public:
    LList(int size=DefaultListSize) { init(); }
    ~LList() { removeall(); } // Destructor
    void clear() { removeall(); init(); }

    void setStart() {
        fence = head; rightcnt += leftcnt;
        leftcnt = 0;
    }

    void setEnd() {
        fence = tail; leftcnt += rightcnt;
        rightcnt = 0;
    }

    void next() {
        // Don't move fence if right empty
        if (fence != tail) {
            fence = fence->next; rightcnt--;
            leftcnt++; }
    }

    int leftLength() const { return leftcnt; }
    int rightLength() const { return rightcnt; }

    bool getValue(Elem &it) const {
        if(rightLength() == 0) return false;
        it = fence->next->element;
        return true; }
    }

    void removeall() { // Return link nodes to free store
        while(head != NULL) {
            fence = head;
            head = head->next;
            delete fence;
        }
    }

    void insert(int i) { // Insert new element
        Link<Elem>* temp = new Link<Elem>;
        temp->element = i;
        temp->next = head;
        head = temp;
    }
}
Insert/Append

// Insert at front of right partition
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
    fence->next =
        new Link<Elem>(item, fence->next);
    if (tail == fence) tail = fence->next;
    rightcnt++;
    return true;
}

// Append Elem to end of the list
template <class Elem>
bool LList<Elem>::append(const Elem& item) {
    tail = tail->next =
        new Link<Elem>(item, NULL);
    rightcnt++;
    return true;
}

Removal

Remove

// Remove and return first Elem in right
// partition
template <class Elem> bool
LList<Elem>::remove(Elem& it) {
    if (fence->next == NULL) return false;
    it = fence->next->element; // Remember val
    // Remember link node
    Link<Elem>* ltemp = fence->next;
    fence->next = ltemp->next; // Remove
    if (tail == ltemp) // Reset tail
        tail = fence;
    delete ltemp;      // Reclaim space
    rightcnt--;
    return true;
}

Prev

// Move fence one step left;
// no change if left is empty
template <class Elem> void
LList<Elem>::prev() {
    Link<Elem>* temp = head;
    if (fence == head) return; // No prev Elem
    while (temp->next!=fence)
        temp=temp->next;
    fence = temp;
    leftcnt--;
    rightcnt++;
}

Setpos

// Set the size of left partition to pos
template <class Elem>
bool LList<Elem>::setPos(int pos) {
    if ((pos < 0) || (pos > rightcnt+leftcnt))
        return false;
    fence = head;
    for(int i=0; i<pos; i++)
        fence = fence->next;
    return true;
}

Comparison of Implementations

Array-Based Lists:
• Insertion and deletion are $\Theta(n)$.
• Prev and direct access are $\Theta(1)$.
• Array must be allocated in advance.
• No overhead if all array positions are full.

Linked Lists:
• Insertion and deletion are $\Theta(1)$.
• Prev and direct access are $\Theta(n)$.
• Space grows with number of elements.
• Every element requires overhead.
Space Comparison

“Break-even” point:

\[ DE = n(P + E); \]
\[ n = \frac{DE}{P + E} \]

\(E\): Space for data value.
\(P\): Space for pointer.
\(D\): Number of elements in array.

Freelists

System new and delete are slow.

// Singly-linked list node with freelists
template <class Elem> class Link {
    private:
        static Link<Elem>* freelist; // Head
    public:
        Elem element;       // Value for this node
        Link* next;         // Pointer to next node
        Link(const Elem& elemval, Link* nextval = NULL) {
            element = elemval;  // Element of the new node
            next = nextval;     // Point to next node
        }
        Link(const Elem& elemval, Link* nextval = NULL) {
            element = elemval;  // Element of the new node
            next = nextval;     // Point to next node
        }
    }

Freelists (2)

template <class Elem>
Link<Elem>* Link<Elem>::freelist = NULL;

Doubly Linked Lists

Simplify insertion and deletion: Add a prev pointer.

// Doubly-linked list link node
template <class Elem> class Link {
    public:
        Elem element;  // Value for this node
        Link* next;    // Pointer to next node
        Link* prev;    // Pointer to previous node
        Link(const Elem& e, Link* prevp = NULL, Link* nextp = NULL) {
            element = e;   // Element of the new node
            prev = prevp;  // Point to previous node
            next = nextp;  // Point to next node
        }
        Link(Link* prevp = NULL, Link* nextp = NULL) {
            prev = prevp;  // Point to previous node
            next = nextp;  // Point to next node
        }
    }

Doubly Linked Insert

// Insert node
// 20 23 12 15
// Insert node
// 20 23 12
// Insert 10
// 20 23 12
// 10
// Insert 5
// 20 23 12
// 5 10
Doubly Linked Insert

// Insert at front of right partition
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
    fence->next =
        new Link<Elem>(item, fence, fence->next);
    if (fence->next->next != NULL)
        fence->next->next->prev = fence->next;
    if (tail == fence)   // Appending new Elem
        tail = fence->next; //   so set tail
    rightcnt++;
    // Added to right
    return true;
}

Doubly Linked Remove

// Remove, return first Elem in right part
template <class Elem>
bool LList<Elem>::remove(Elem& it) {
    if (fence->next == NULL) return false;
    it = fence->next->element;
    Link<Elem>* ltemp = fence->next;
    if (ltemp->next != NULL)
        ltemp->next->prev = fence;
    else tail = fence;         // Reset tail
    fence->next = ltemp->next; // Remove
    delete ltemp;    // Reclaim space
    rightcnt--;      // Removed from right
    return true;
}

Dictionary

Often want to insert records, delete records, search for records.

Required concepts:
• Search key: Describe what we are looking for
• Key comparison
  – Equality: sequential search
  – Relative order: sorting
• Record comparison

Comparator Class

How do we generalize comparison?
• Use ==, <, >: Disastrous
• Overload ==, <, >: Disastrous
• Define a function with a standard name
  – Implied obligation
  – Breaks down with multiple key fields/indices for same object
• Pass in a function
  – Explicit obligation
  – Function parameter
  – Template parameter

Comparator Example

class intintCompare {
public:
    static bool lt(int x, int y) {
        return x < y;
    }
    static bool eq(int x, int y) {
        return x == y;
    }
    static bool gt(int x, int y) {
        return x > y;
    }
};
Comparator Example (2)

```cpp
class PayRoll {
public:
    int ID;
    char* name;
};
class IDCompare {
public:
    static bool lt(Payroll& x, Payroll& y)
    { return x.ID < y.ID; }
};
class NameCompare {
public:
    static bool lt(Payroll& x, Payroll& y)
    { return strcmp(x.name, y.name) < 0; }
};
```

Dictionary ADT

```cpp
// The Dictionary abstract class.
template <class Key, class Elem,
    class KEComp, class EEComp>
class Dictionary {
public:
    virtual void clear() = 0;
    virtual bool insert(const Elem&) = 0;
    virtual bool remove(const Key&, Elem&) = 0;
    virtual bool removeAny(Elem&) = 0;
    virtual bool find(const Key&, Elem&) const = 0;
    virtual int size() = 0;
};
```

Unsorted List Dictionary

```cpp
template <class Key, class Elem,
    class KEComp, class EEComp>
class UALdict : public Dictionary<Key,Elem,KEComp,EEComp> {
private: AList<Elem>* list;
public:
    bool remove(const Key& K, Elem& e) {
        for(list->setStart(); list->getValue(e);
            list->next())
            if (KEComp::eq(K, e)) {
                list->remove(e);
                return true;
            }
        return false;
    }
};
```

Stacks

LIFO: Last In, First Out.

Restricted form of list: Insert and remove only at front of list.

Notation:
- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.

Stack ADT

```cpp
// Stack abstract class
template <class Elem> class Stack {
public:
    // Reinitialize the stack
    virtual void clear() = 0;
    // Push an element onto the top of the stack.
    virtual bool push(const Elem&) = 0;
    // Remove the element at the top of the stack.
    virtual bool pop(Elem&) = 0;
    // Get a copy of the top element in the stack
    virtual bool topValue(Elem&) const = 0;
    // Return the number of elements in the stack.
    virtual int length() const = 0;
};
```

Array-Based Stack

```cpp
// Array-based stack implementation
private:
    int size; // Maximum size of stack
    int top; // Index for top element
    Elem *listArray; // Array holding elements

Issues:
- Which end is the top?
- Where does “top” point to?
- What is the cost of the operations?
Linked Stack

// Linked stack implementation
private:
    Link<Elem>* top; // Pointer to first elem
    int size;        // Count number of elems

What is the cost of the operations?

How do space requirements compare to the array-based stack implementation?

Queues

FIFO: First in, First Out

Restricted form of list: Insert at one end, remove from the other.

Notation:
- Insert: Enqueue
- Delete: Dequeue
- First element: Front
- Last element: Rear

Queue Implementation (1)

Queue Implementation (2)

Binary Trees

A binary tree is made up of a finite set of nodes that is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root.

Binary Tree Example

Notation: Node, children, edge, parent, ancestor, descendant, path, depth, height, level, leaf node, internal node, subtree.
Full and Complete Binary Trees

**Full binary tree:** Each node is either a leaf or internal node with exactly two non-empty children.

**Complete binary tree:** If the height of the tree is $d$, then all leaves except possibly level $d$ are completely full. The bottom level has all nodes to the left side.

---

Full Binary Tree Theorem (1)

**Theorem:** The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

**Proof (by Mathematical Induction):**

**Base case:** A full binary tree with 1 internal node must have two leaf nodes.

**Induction Hypothesis:** Assume any full binary tree $T$ containing $n-1$ internal nodes has $n$ leaves.

**Induction Step:** Given tree $T$ with $n$ internal nodes, pick internal node $I$ with two leaf children. Remove $I$'s children, call resulting tree $T'$. By induction hypothesis, $T'$ is a full binary tree with $n$ leaves.

Restore $I$'s two children. The number of internal nodes has now gone up by 1 to reach $n$. The number of leaves has also gone up by 1.

---

Full Binary Tree Theorem (2)

**Induction Step:** Given tree $T$ with $n$ internal nodes, pick internal node $I$ with two leaf children. Remove $I$'s children, call resulting tree $T'$. By induction hypothesis, $T'$ is a full binary tree with $n$ leaves.

Restore $I$'s two children. The number of internal nodes has now gone up by 1 to reach $n$. The number of leaves has also gone up by 1.

---

Full Binary Tree Corollary

**Theorem:** The number of null pointers in a non-empty tree is one more than the number of nodes in the tree.

**Proof:** Replace all null pointers with a pointer to an empty leaf node. This is a full binary tree.

---

Binary Tree Node Class (1)

```cpp
// Binary tree node class
template <class Elem>
class BinNodePtr : public BinNode<Elem> {
private:
    Elem it;         // The node's value
    BinNodePtr* lc;  // Pointer to left child
    BinNodePtr* rc;  // Pointer to right child
public:
    BinNodePtr() { lc = rc = NULL; }
    BinNodePtr(Elem e, BinNodePtr* l = NULL, BinNodePtr* r = NULL) {
        it = e; lc = l; rc = r; }
    Elem& val() { return it; }
    void setVal(const Elem& e) { it = e; }
    inline BinNode<Elem>* left() const
    { return lc; }
    void setLeft(BinNode<Elem>* b) { lc = (BinNodePtr*)b; }
    inline BinNode<Elem>* right() const
    { return rc; }
    void setRight(BinNode<Elem>* b) { rc = (BinNodePtr*)b; }
    bool isLeaf()
    { return (lc == NULL) && (rc == NULL); }
};
```

---

Binary Tree Node Class (2)

```cpp
Elem& val() { return it; }
void setVal(const Elem& e) { it = e; }
inline BinNode<Elem>* left() const
{ return lc; }
void setLeft(BinNode<Elem>* b) { lc = (BinNodePtr*)b; }
inline BinNode<Elem>* right() const
{ return rc; }
void setRight(BinNode<Elem>* b) { rc = (BinNodePtr*)b; }
bool isLeaf()
{ return (lc == NULL) && (rc == NULL); }
```
Traversals (1)

Any process for visiting the nodes in some order is called a traversal.

Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes.

Traversals (2)

- Preorder traversal: Visit each node before visiting its children.
- Postorder traversal: Visit each node after visiting its children.
- Inorder traversal: Visit the left subtree, then the node, then the right subtree.

Traversals (3)

template <class Elem> // Good implementation
void preorder(BinNode<Elem>* subroot) {
    if (subroot == NULL) return; // Empty
    visit(subroot); // Perform some action
    preorder(subroot->left());
    preorder(subroot->right());
}

template <class Elem> // Bad implementation
void preorder2(BinNode<Elem>* subroot) {
    visit(subroot); // Perform some action
    if (subroot->left() != NULL) 
        preorder2(subroot->left());
    if (subroot->right() != NULL) 
        preorder2(subroot->right());
}

Traversal Example

// Return the number of nodes in the tree
template <class Elem>
int count(BinNode<Elem>* subroot) {
    if (subroot == NULL)
        return 0; // Nothing to count
    return 1 + count(subroot->left())
        + count(subroot->right());
}

Binary Tree Implementation (1)

Binary Tree Implementation (2)
Union Implementation (1)

```cpp
enum Nodetype {leaf, internal};

class VarBinNode { // Generic node class
c-public:
Nodetype mytype; // Store type for node
union {
    struct { // internal node
        VarBinNode* left; // Left child
        VarBinNode* right; // Right child
        Operator opx; // Value
    } intl;
    Operand var; // Leaf: Value only
};
```

Union Implementation (2)

```cpp
// Leaf constructor
VarBinNode(const Operand& val) {
    mytype = leaf; var = val;
}

// Internal node constructor
VarBinNode(const Operator& op,
        VarBinNode* l, VarBinNode* r) {
    mytype = internal; intl.opx = op;
    intl.left = l; intl.right = r;
}

bool isLeaf() { return mytype == leaf; }
VarBinNode* leftchild() {
    return intl.left;
}
VarBinNode* rightchild() {
    return intl.right;
}
```

Union Implementation (3)

```cpp
// Preorder traversal
void traverse(VarBinNode* subroot) {
    if (subroot == NULL) return;
    if (subroot->isLeaf())
        cout << "Leaf: " << subroot->var << ":n;";
    else {
        cout << "Internal: " << subroot->intl.opx << ":n;"
        traverse(subroot->leftchild());
        traverse(subroot->rightchild());
    }
}
```

Inheritance (1)

```cpp
class VarBinNode { // Abstract base class
    public:
        virtual bool isLeaf() = 0;
};

class LeafNode : public VarBinNode { // Leaf
    private:
        Operand var; // Operand value
    public:
        LeafNode(const Operand& val) {
            var = val;
        }

        bool isLeaf() { return true; }

        Operand value() { return var; }
};
```

Inheritance (2)

```cpp
// Internal node
class IntlNode : public VarBinNode {
    private:
        VarBinNode* left; // Left child
        VarBinNode* right; // Right child
        Operator opx; // Operator value
    public:
        IntlNode(const Operator& op,
                VarBinNode* l, VarBinNode* r) {
            opx = op; left = l; right = r;
        }

        bool isLeaf() { return false; }

        VarBinNode* leftchild() { return left; }
        VarBinNode* rightchild() { return right; }

        Operator value() { return opx; }
};
```

Inheritance (3)

```cpp
// Preorder traversal
void traverse(VarBinNode *subroot) {
    if (subroot == NULL) return; // Empty
    if (subroot->isLeaf()) // Do leaf node
        cout << "Leaf: " << ((LeafNode *)subroot)->value() << ":n;"
    else if (subroot->isLeaf()) // Do leaf node
        cout << "Leaf: " << ((LeafNode *)subroot)->value() << ":n;"
    else {
        cout << "Internal: " << ((IntlNode *)subroot)->value() << ":n;"
        traverse(((IntlNode *)subroot)->leftchild());
        traverse(((IntlNode *)subroot)->rightchild());
    }
}
class VarBinNode {   // Abstract base class
public:
    virtual bool isLeaf() = 0;
    virtual void trav() = 0;
};
class LeafNode : public VarBinNode { // Leaf
private:
    Operand var;              // Operand value
public:
    LeafNode(const Operand& val)
    { var = val; } // Constructor
    bool isLeaf() { return true; }
    Operand value() { return var; }
    void trav() { cout << "Leaf: " << value() << endl; }
};

class IntlNode : public VarBinNode { // Internal
private:
    VarBinNode* lc;   // Left child
    VarBinNode* rc;   // Right child
    Operator opx;     // Operator value
public:
    IntlNode(const Operator& op,
              VarBinNode* l, VarBinNode* r)
        { opx = op; lc = l; rc = r; }
    bool isLeaf() { return false; }
    VarBinNode* left() { return lc; }
    VarBinNode* right() { return rc; }
    Operator value() { return opx; }
    void trav() {
        cout << "Internal: " << value() << endl;
        if (left() != NULL) left()->trav();
        if (right() != NULL) right()->trav();
    }
};

// Preorder traversal
void traverse(VarBinNode *root) {
    if (root != NULL)
        root->trav();
}

// From the Full Binary Tree Theorem:
// Half of the pointers are null.
// If leaves store only data, then overhead depends on whether the tree is full.
// Ex: All nodes the same, with two pointers to children:
// Total space required is (2p + d)n
// Overhead: 2pn
// If p = d, this means 2p(2p + d) = 2/3 overhead.

Eliminate pointers from the leaf nodes:
\[
\frac{n/2(2p)}{n/2(2p) + dn} = \frac{p}{p + d}
\]
This is 1/2 if p = d.

2p(2p + d) if data only at leaves = 2/3 overhead.

Note that some method is needed to distinguish leaves from internal nodes.

Array Implementation

<table>
<thead>
<tr>
<th>Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Left Child</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Right Child</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Left Sibling</td>
<td>--</td>
<td>1</td>
<td>--</td>
<td>3</td>
<td>--</td>
<td>5</td>
<td>--</td>
<td>7</td>
<td>--</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right Sibling</td>
<td>--</td>
<td>2</td>
<td>--</td>
<td>4</td>
<td>--</td>
<td>6</td>
<td>--</td>
<td>8</td>
<td>--</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21
Array Implementation (1)

Parent \( (r) = \)
Leftchild\( (r) = \)
Rightchild\( (r) = \)
Leftsibling\( (r) = \)
Rightsibling\( (r) = \)

Binary Search Trees

BST Property: All elements stored in the left subtree of a node with value \( K \) have values < \( K \). All elements stored in the right subtree of a node with value \( K \) have values >= \( K \).

BST ADT(1)

// BST implementation for the Dictionary ADT
template <class Key, class Elem,
class KEComp, class EEComp>
class BST : public Dictionary<Key, Elem,
KEComp, EEComp> {
private:
BinNode<Elem>* root; // Root of the BST
int nodecount; // Number of nodes
void clearhelp(BinNode<Elem>*);
BinNode<Elem>*
inserthelp(BinNode<Elem>*, const Elem&);
BinNode<Elem>*
deletemin(BinNode<Elem>*, BinNode<Elem>**)&;
BinNode<Elem>*
removehelp(BinNode<Elem>*,
const Key&, BinNode<Elem>**)&;
bool findhelp(BinNode<Elem>*, const Key&, Elem&); // BST ADT(2)
void printhelp(BinNode<Elem>*, int) const;

BST ADT(2)

public:
BST() { root = NULL; nodecount = 0; }
~BST() { clearhelp(root); }
void clear() { clearhelp(root); root = NULL;
nodecount = 0; }
bool insert(const Elem& e) {
root = inserthelp(root, e);
nodecount++;
return true; }
bool remove(const Key& K, Elem& e) {
BinNode<Elem>* t = NULL;
root = removehelp(root, K, t);
if (t == NULL) return false;
e = t->val();
nodecount--;
delete t;
return true; }

BST ADT(3)

bool removeAny(Elem& e) { // Delete min value
if (root == NULL) return false; // Empty
BinNode<Elem>* t = root;
root = deletemin(root, t);
delete t;
nodecount--;
return true; }
bool find(const Key& K, Elem& e) const {
return findhelp(root, K, e); }
int size() { return nodecount; }
void print() const { // BST Search
template <class Key, class Elem,
class KEComp, class EEComp>
bool BST<Key, Elem, KEComp, EEComp>::
findhelp(BinNode<Elem>* subroot,
const Key& K, Elem& e) const {
if (subroot == NULL) return false;
else if (KEComp::lt(K, subroot->val()))
return findhelp(subroot->left(), K, e);
else if (KEComp::gt(K, subroot->val()))
return findhelp(subroot->right(), K, e);
else { e = subroot->val(); return true; }
}
**BST Insert (1)**

```
template <class Key, class Elem, class KEComp, class EEComp>
    BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
    inserthelp(BinNode<Elem>* subroot, const Elem & val) {
        if (subroot == NULL) // Empty: create node
            return new BinNodePtr<Elem>(val,NULL,NULL);
        if (EEComp::lt(val, subroot->val()))
            subroot->setLeft(inserthelp(subroot->left(), val));
        else subroot->setRight(inserthelp(subroot->right(), val));
        return subroot;
    }
```

**BST Insert (2)**

```
template <class Key, class Elem, class KEComp, class EEComp>
    BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
    inserthelp(BinNode<Elem>* subroot, const Elem & val) {
        if (subroot == NULL) // Empty: create node
            return new BinNodePtr<Elem>(val,NULL,NULL);
        if (EEComp::lt(val, subroot->val()))
            subroot->setLeft(inserthelp(subroot->left(), val));
        else subroot->setRight(inserthelp(subroot->right(), val));
        return subroot;
    }
```

**Remove Minimum Value**

```
template <class Key, class Elem, class KEComp, class EEComp>
    BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
    deletemin(BinNode<Elem>* subroot, BinNode<Elem>*& min) {
        if (subroot->left() == NULL) {
            min = subroot;
            return subroot->right();
        }
        else { // Continue left
            subroot->setLeft(deletemin(subroot->left(), min));
            return subroot;
        }
    }
```

**BST Remove (1)**

```
template <class Key, class Elem, class KEComp, class EEComp>
    BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
    removehelp(BinNode<Elem>* subroot, const Key & K, BinNode<Elem>*& t) {
        if (subroot == NULL) return NULL;
        else if (KEComp::lt(K, subroot->val()))
            subroot->setLeft(removehelp(subroot->left(), K, t));
        else if (KEComp::gt(K, subroot->val()))
            subroot->setRight(removehelp(subroot->right(), K, t));
        else { // Found it: remove it
            BinNode<Elem>* temp;
            t = subroot;
            if (subroot->left() == NULL)
                subroot = subroot->right();
            else if (subroot->right() == NULL)
                subroot = subroot->left();
            else { // Both children are non-empty
                subroot->setRight(deletemin(subroot->right(), temp));
                Elem te = subroot->val();
                subroot->setVal(temp->val());
                temp->setVal(te);
                t = temp;
            }
            return subroot;
        }
    }
```

**BST Remove (2)**

```
template <class Key, class Elem, class KEComp, class EEComp>
    BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
    removehelp(BinNode<Elem>* subroot, const Key & K, BinNode<Elem>*& t) {
        if (subroot == NULL) return NULL;
        else if (KEComp::lt(K, subroot->val()))
            subroot->setLeft(removehelp(subroot->left(), K, t));
        else if (KEComp::gt(K, subroot->val()))
            subroot->setRight(removehelp(subroot->right(), K, t));
        else { // Found it: remove it
            BinNode<Elem>* temp;
            t = subroot;
            if (subroot->left() == NULL)
                subroot = subroot->right();
            else if (subroot->right() == NULL)
                subroot = subroot->left();
            else { // Both children are non-empty
                subroot->setRight(deletemin(subroot->right(), temp));
                Elem te = subroot->val();
                subroot->setVal(temp->val());
                temp->setVal(te);
                t = temp;
            }
            return subroot;
        }
    }
```
Cost of BST Operations

- **Find:**
- **Insert:**
- **Delete:**

Heap ADT

```cpp
template<class Elem, class Comp>
class maxheap{
private:
    Elem* Heap;   // Pointer to the heap array
    int size;     // Maximum size of the heap
    int n;        // Number of elems now in heap

    void siftdown(int pos) { // Put element in place
        while (!isLeaf(pos)) {
            int j = leftchild(pos);
            int rc = rightchild(pos);
            if ((rc<n) && Comp::lt(Heap[j],Heap[rc]))
                j = rc;
            if (!Comp::lt(Heap[pos], Heap[j])) return;
            swap(Heap, pos, j);
            pos = j;
        }
    }
public:
    maxheap(Elem* h, int num, int max);
    int heapsize() const;
    bool isLeaf(int pos) const;
    int leftchild(int pos) const;
    int rightchild(int pos) const;
    int parent(int pos) const;
    bool insert(const Elem&);
    bool removemax(Elem&);
    bool remove(int, Elem&);
    void buildHeap();
};
```

Building the Heap

(a) (4-2) (4-1) (2-1) (5-2) (5-4) (6-3) (6-5) (7-5) (7-6) (b) (5-2), (7-3), (7-1), (6-1)

Siftdown (1)

- Work from high end of array to low end.
- Call siftdown for each item.
- Don’t need to call siftdown on leaf nodes.

```cpp
template <class Elem, class Comp>
void maxheap<Elem,Comp>::siftdown(int pos) {
    int j = leftchild(pos);
    int rc = rightchild(pos);
    if ((rc<n) && Comp::lt(Heap[j],Heap[rc]))
        j = rc;
    if (!Comp::lt(Heap[pos], Heap[j])) return;
    swap(Heap, pos, j);  
    pos = j;
}
```

Siftdown (2)
Buildheap Cost

Cost for heap construction:

\[ \log n \sum_{i=1}^{n/2} (i-1) \approx n. \]

Remove Max Value

```cpp
template <class Elem, class Comp>
bool maxheap<Elem, Comp>:::
    removemax(Elem &it) { 
        if (n == 0) return false; // Heap is empty
        swap(Heap, 0, --n);   // Swap max with end
        if (n != 0) siftdown(0); // Return max value
        it = Heap[n];
        return true;
    }
```

Priority Queues (1)

A priority queue stores objects, and on request releases the object with greatest value.

Example: Scheduling jobs in a multi-tasking operating system.

The priority of a job may change, requiring some reordering of the jobs.

Implementation: Use a heap to store the priority queue.

Priority Queues (2)

To support priority reordering, delete and re-insert. Need to know index for the object in question.

```cpp
template <class Elem, class Comp>
bool maxheap<Elem, Comp>::remove(int pos, 
    Elem &it) {
    if ((pos < 0) || (pos >= n)) return false;
    swap(Heap, pos, --n);
    while ((pos != 0) && (Comp::gt(Heap[pos],
        Heap[parent(pos)])))
        swap(Heap, pos, parent(pos));
    siftdown(pos);
    it = Heap[n];
    return true;
}
```

Huffman Coding Trees

ASCII codes: 8 bits per character.

• Fixed-length coding.

Can take advantage of relative frequency of letters to save space.

• Variable-length coding

<table>
<thead>
<tr>
<th>Z</th>
<th>K</th>
<th>F</th>
<th>C</th>
<th>U</th>
<th>D</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>24</td>
<td>32</td>
<td>37</td>
<td>42</td>
<td>42</td>
<td>120</td>
</tr>
</tbody>
</table>

Build the tree with minimum external path weight.

Huffman Tree Construction (1)

```
Step 1
  2
-- 7
  3
Step 2
  2
-- 7
  3
Step 3
  2
-- 7
  3
```

```
Step 1
  2 7 24 32 37 42 42 120
Step 2
  2 7 32 37 42 42 120
Step 3
  2 7 32 120
```
Assigning Codes

<table>
<thead>
<tr>
<th>Letter</th>
<th>Freq</th>
<th>Code</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Coding and Decoding

A set of codes is said to meet the prefix property if no code in the set is the prefix of another.

Code for DEED:

Decode 1011001110111101:

Expected cost per letter:

General Trees

General Tree Node

```cpp
// General tree node ADT
template <class Elem> class GTNode {
public:
    GTNode(const Elem&); // Constructor
    ~GTNode();           // Destructor
    Elem value();        // Return value
    bool isLeaf();       // TRUE if is a leaf
    GTNode* parent();    // Return parent
    GTNode* leftmost_child(); // First child
    GTNode* right_sibling();  // Right sibling
    void setValue(Elem&);     // Set value
    void insert_first(GTNode<Elem>* n);
    void insert_next(GTNode<Elem>* n);
    void remove_first(); // Remove first child
    void remove_next(); // Remove sibling
};
```

General Tree Traversal

```cpp
template <class Elem>
void GenTree<Elem>::printhelp(GTNode<Elem>* subroot) {
    if (subroot->isLeaf()) cout << "Leaf: ";
    else cout << "Internal: ";
    cout << subroot->value() << "\n";
    for (GTNode<Elem>* temp = subroot->leftmost_child(); temp != NULL;
           temp = temp->right_sibling())
        printhelp(temp);
}
```
Parent Pointer Implementation

The parent pointer representation is good for answering:
- Are two elements in the same tree?

```cpp
bool Gentree::differ(int a, int b) {
  int root1 = FIND(a);   // Find root for a
  int root2 = FIND(b);   // Find root for b
  return root1 != root2; // Compare roots
}
```

Union/Find

```cpp
void Gentree::UNION(int a, int b) {
  int root1 = FIND(a);   // Find root for a
  int root2 = FIND(b); // Find root for b
  if (root1 != root2) array[root2] = root1;
}
```

```cpp
int Gentree::FIND(int curr) const {
  while (array[curr] != ROOT) curr = array[curr];
  return curr;  // At root
}
```

Want to keep the depth small.

Weighted union rule: Join the tree with fewer nodes to the tree with more nodes.

Equivalence Class Problem

```cpp
// Return TRUE if nodes in different trees
bool Gentree::differ(int a, int b) {
  int root1 = FIND(a);   // Find root for a
  int root2 = FIND(b); // Find root for b
  return root1 != root2; // Compare roots
}
```

Equiv Class Processing (1)

Equiv Class Processing (2)

Path Compression

```cpp
int Gentree::FIND(int curr) const {
  if (array[curr] == ROOT) return curr;
  return array[curr] = FIND(array[curr]);
}
```
Lists of Children

<table>
<thead>
<tr>
<th>Index</th>
<th>Val</th>
<th>Par</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Leftmost Child/Right Sibling (1)

Leftmost Child/Right Sibling (2)

Linked Implementations (1)

Linked Implementations (2)

Converting to a Binary Tree

Left child/right sibling representation essentially stores a binary tree.

Use this process to convert any general tree to a binary tree.

A forest is a collection of one or more general trees.
Sequential Implementations (1)

List node values in the order they would be visited by a preorder traversal.
Saves space, but allows only sequential access.
Need to retain tree structure for reconstruction.
Example: For binary trees, use a symbol to mark null links.
AB/D//CEG///FH///

Sequential Implementations (2)

Example: For full binary trees, mark nodes as leaf or internal.
A'B'/DC'E'G/F'HI

Example: For general trees, mark the end of each subtree.
RAC(D(E)(BF)))

Sorting

Each record contains a field called the key.
  – Linear order: comparison.

Measures of cost:
  – Comparisons
  – Swaps

Insertion Sort (1)

```
template <class Elem, class Comp>
void inssort(Elem A[], int n) {
  for (int i=1; i<n; i++) {
    for (int j=i; (j>0) &&
    (Comp::lt(A[j], A[j-1])); j--) {
      swap(A, j, j-1);
    }
  }
}
```

Best Case:
Worst Case:
Average Case:

Insertion Sort (2)

Bubble Sort (1)

```
template <class Elem, class Comp>
void bubble_sort(Elem A[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++) {
      if (A[j] > A[j+1]) {
        swap(A, j, j+1);
      }
    }
  }
}
```

```
template <class Elem, class Comp>
void bubble_sort(Elem A[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++) {
      if (A[j] > A[j+1]) {
        swap(A, j, j+1);
      }
    }
  }
}
```

```
template <class Elem, class Comp>
void bubble_sort(Elem A[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++) {
      if (A[j] > A[j+1]) {
        swap(A, j, j+1);
      }
    }
  }
}
```

```
template <class Elem, class Comp>
void bubble_sort(Elem A[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++) {
      if (A[j] > A[j+1]) {
        swap(A, j, j+1);
      }
    }
  }
}
```
Bubble Sort (2)

```cpp
template <class Elem, class Comp>
void bubsort(Elem A[], int n) {
    for (int i=0; i<n-1; i++)
        for (int j=n-1; j>i; j--)
            if (Comp::lt(A[j], A[j-1]))
                swap(A, j, j-1);
}
```

Best Case:
Worst Case:
Average Case:

Selection Sort (1)

```cpp
template <class Elem, class Comp>
void selsort(Elem A[], int n) {
    for (int i=0; i<n-1; i++) {
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--)
            if (Comp::lt(A[j], A[lowindex]))
                lowindex = j; // Put it in place
        swap(A, i, lowindex);
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>

Best Case:
Worst Case:
Average Case:

Selection Sort (2)

```cpp
template <class Elem, class Comp>
void selsort(Elem A[], int n) {
    for (int i=0; i<n-1; i++) {
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--)
            if (Comp::lt(A[j], A[lowindex]))
                lowindex = j; // Put it in place
        swap(A, i, lowindex);
    }
}
```

Best Case:
Worst Case:
Average Case:

Pointer Swapping

Exchange Sorting

All of the sorts so far rely on exchanges of adjacent records.

What is the average number of exchanges required?
- There are \( n! \) permutations
- Consider permutation \( X \) and its reverse, \( X' \)
- Together, every pair requires \( n(n-1)/2 \) exchanges.

Summary

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Bubble</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparisons</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Case</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>Average Case</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>Worst Case</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td><strong>Swaps</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Case</td>
<td>0</td>
<td>0</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Average Case</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Worst Case</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>
Shellsort

// Modified version of Insertion Sort
template <class Elem, class Comp>
void inssort2(Elem A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr)
        for (int j=i; 
            (j>=incr) &&
            (Comp::lt(A[j], A[j-incr])); j-=incr)
            swap(A, j, j-incr);
}

template <class Elem, class Comp>
void shellsort(Elem A[], int n) { // Shellsort
    for (int i=n/2; i>2; i/=2)  // For each incr
        for (int j=0; j<i; j++)   // Sort sublists
            inssort2<Elem,Comp>(&A[j], n-j, i);
    inssort2<Elem,Comp>(A, n, 1);
}

Quicksort

template <class Elem, class Comp>
void qsort(Elem A[], int i, int j) {
    if (j <= i) return; // List too small
    int pivotindex = findpivot(A, i, j);
    swap(A, pivotindex, j);  // Put pivot at end
    // k will be first position on right side
    int k =
        partition<Elem,Comp>(A, i-1, j, A[j]);
    qsort<Elem,Comp>(A, i, k-1);
    qsort<Elem,Comp>(A, k+1, j);
}

template <class Elem, class Comp>
int findpivot(Elem A[], int i, int j){
    return (i+j)/2;
}

Quicksort Partition

template <class Elem, class Comp>
int partition(Elem A[], int l, int r, 
        Elem &pivot) {
        do { // Move the bounds in until they meet
            while (Comp::lt(A[++l], pivot));
            while ((r != 0) && Comp::gt(A[--r], pivot));
            swap(A, l, r); // Swap out-of-place values
        } while (l < r); // Stop when they cross
        swap(A, l, r); // Reverse last swap
        return l; // Return first pos on right
}

The cost for partition is \( \Theta(n) \).
Cost of Quicksort

Best case: Always partition in half.
Worst case: Bad partition.
Average case:
\[ T(n) = n + 1 + \frac{1}{n-1} \sum_{k=1}^{n-1} (T(k) + T(n-k)) \]

Optimizations for Quicksort:
- Better Pivot
- Better algorithm for small sublists
- Eliminate recursion

Mergesort

List mergesort(List inlist) {
    if (inlist.length() <= 1) return inlist;
    List l1 = half of the items from inlist;
    List l2 = other half of items from inlist;
    return merge(mergesort(l1), mergesort(l2));
}

Mergesort Implementation

template <class Elem, class Comp>
void mergesort(Elem A[], Elem temp[], int left, int right) {
    int mid = (left+right)/2;
    if (left == right) return;
    mergesort<Elem,Comp>(A, temp, left, mid);
    mergesort<Elem,Comp>(A, temp, mid+1, right);
    for (int i=left; i<=right; i++) // Copy
        temp[i] = A[i];
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1) // Left exhausted
            A[curr] = temp[i2++];
        else if (i2 > right) // Right exhausted
            A[curr] = temp[i1++];
        else if (Comp::lt(temp[i1], temp[i2]))
            A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
}

Optimized Mergesort

template <class Elem, class Comp>
void mergesort(Elem A[], Elem temp[], int left, int right) {
    if ((right-left) <= THRESHOLD) {
        inssort<Elem,Comp>(&A[left],right-left+1);
        return;
    }
    int i, j, k, mid = (left+right)/2;
    if (left == right) return;
    mergesort<Elem,Comp>(A, temp, left, mid);
    mergesort<Elem,Comp>(A, temp, mid+1, right);
    for (i=mid; i>=left; i--) temp[i] = A[i];
    for (j=1; j<=right-mid; j++)
        temp[right-j+1] = A[j+mid];
    for (i=left, j=right, k=left; k<=right; k++)
        if (temp[i] < temp[j]) A[k] = temp[i++];
        else A[k] = temp[j--];
}

Mergesort Cost

Mergesort cost:
Mergesort is also good for sorting linked lists.
Mergesort requires twice the space.

Heapsort

template <class Elem, class Comp>
void heapsort(Elem A[], int n) { // Heapsort
    Elem mval;
    maxheap<Elem,Comp> H(A, n, n);
    for (int i=0; i<n; i++) // Now sort
        H.remove(maxval); // Put max at end
}

Use a max-heap, so that elements end up sorted within the array.
Cost of heapsort:
Cost of finding K largest elements:
Heapsort Example (1)

Original Numbers

73 6 57 80 42 83 72 48 85

Build Heap

85 83 72 73 42 57 6 48 80

Remove 85

57 42 80 73 6 48 80

Remove 73

57 42 80 6 48 80

Remove 57

42 80 6 48 80

Heapsort Example (2)

Remove 85

83 57 60 42 57 6 48 80

Remove 60

83 57 42 57 6 48 80

Remove 57

83 42 57 6 48 80

Remove 42

83 57 6 48 80

Remove 57

83 6 48 80

Remove 42

83 6 48 80

Binsort (1)

A simple, efficient sort:

for (i=0; i<n; i++)
B[A[i]] = A[i];

Ways to generalize:

– Make each bin the head of a list.
– Allow more keys than records.

Binsort (2)

template <class Elem>
void binsort(Elem A[], int n) {
List<Elem> B[MaxKeyValue];
Elem item;
for (i=0; i<n; i++) B[A[i]].append(A[i]);
for (i=0; i<MaxKeyValue; i++)
for (B[i].setStart();
B[i].getValue(item); B[i].next())
output(item);
}

Cost:

Radix Sort (1)

Initial List: 27 91 1 97 17 23 84 38 72 5 67 25

First pass (right to left)

27 91 1 97 17 23 84 38 72 5 67 25

Second pass (left to right)

1 91 97 27 23 17 84 38 72 5 67 25

Result of first pass: 91 1 97 27 23 17 84 38 72 5 67 25

Result of second pass: 1 5 17 23 27 38 42 67 72 84 91 97

Radix Sort (2)

template <class Elem, class Comp>
void radix(Elem A[], Elem B[],
int n, int k, int r, int cnt[]) {
// cnt[i] stores # of records in bin[i]
int j;
for (int i=0, rtok=1; i<k; i++, rtok*=r) {
for (j=0; j<r; j++) cnt[j] = 0;
// Count # of records for each bin
for(j=0; j<n; j++) cnt[(A[j]/rtok)%r]++;
// cnt[j] will be last slot of bin j.
for (j=0; j<r; j++)
for (j=n-1, j>=0; j--) \nB[--cnt[(A[j]/rtok)%r]] = A[j];
for (j=0; j<n; j++) A[j] = B[j];
}}
Radix Sort Example

Initial Input: Array A

First pass results for Counting sort x 1:

Second pass results for Counting sort x 2:

End of pass 1: Array A

End of pass 2: Array A

Radix Sort Cost

Cost: \( \Theta(nk + rk) \)

How do \( n \), \( k \), and \( r \) relate?

If key range is small, then this can be \( \Theta(n) \).

If there are \( n \) distinct keys, then the length of a key must be at least \( \log n \).

– Thus, Radix Sort is \( \Theta(n \log n) \) in general case

Empirical Comparison (1)

<table>
<thead>
<tr>
<th>Sort</th>
<th>TB</th>
<th>TQ</th>
<th>TK</th>
<th>TDK</th>
<th>TLM</th>
<th>Up</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>0.11</td>
<td>0.03</td>
<td>2.06</td>
<td>502.3</td>
<td>4794.7</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>Radix</td>
<td>0.11</td>
<td>0.05</td>
<td>0.18</td>
<td>1066.1</td>
<td>1200.0</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>Selection</td>
<td>0.11</td>
<td>0.07</td>
<td>5.07</td>
<td>565.7</td>
<td>8855.7</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>Shell</td>
<td>0.11</td>
<td>0.08</td>
<td>1.59</td>
<td>2.0</td>
<td>170.3</td>
<td>8855.7</td>
<td>2.8</td>
</tr>
<tr>
<td>Quick</td>
<td>0.11</td>
<td>0.09</td>
<td>3.09</td>
<td>3.0</td>
<td>180.1</td>
<td>2800.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Quick/100</td>
<td>0.11</td>
<td>0.05</td>
<td>1.35</td>
<td>5.0</td>
<td>400.0</td>
<td>1.7</td>
<td>2.2</td>
</tr>
<tr>
<td>Merge</td>
<td>0.09</td>
<td>0.16</td>
<td>0.77</td>
<td>5.0</td>
<td>44.3</td>
<td>550.0</td>
<td>1.7</td>
</tr>
<tr>
<td>Merge/10</td>
<td>0.09</td>
<td>0.21</td>
<td>0.61</td>
<td>0.5</td>
<td>135.1</td>
<td>1400.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Merge/O</td>
<td>0.09</td>
<td>0.22</td>
<td>0.63</td>
<td>4.6</td>
<td>65.0</td>
<td>990.0</td>
<td>2.7</td>
</tr>
<tr>
<td>Heaps</td>
<td>0.08</td>
<td>0.24</td>
<td>0.36</td>
<td>4.0</td>
<td>24.0</td>
<td>1400.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Radix/A</td>
<td>0.08</td>
<td>0.24</td>
<td>0.36</td>
<td>4.0</td>
<td>24.0</td>
<td>1400.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Empirical Comparison (2)

<table>
<thead>
<tr>
<th>Sort</th>
<th>TB</th>
<th>TQ</th>
<th>TK</th>
<th>TDK</th>
<th>TLM</th>
<th>Up</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>0.11</td>
<td>0.03</td>
<td>2.06</td>
<td>502.3</td>
<td>4794.7</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>Radix</td>
<td>0.11</td>
<td>0.05</td>
<td>0.18</td>
<td>1066.1</td>
<td>1200.0</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>Selection</td>
<td>0.11</td>
<td>0.07</td>
<td>5.07</td>
<td>565.7</td>
<td>8855.7</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>Shell</td>
<td>0.11</td>
<td>0.08</td>
<td>1.59</td>
<td>2.0</td>
<td>170.3</td>
<td>8855.7</td>
<td>2.8</td>
</tr>
<tr>
<td>Quick</td>
<td>0.11</td>
<td>0.09</td>
<td>3.09</td>
<td>3.0</td>
<td>180.1</td>
<td>2800.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Quick/100</td>
<td>0.11</td>
<td>0.05</td>
<td>1.35</td>
<td>5.0</td>
<td>400.0</td>
<td>1.7</td>
<td>2.2</td>
</tr>
<tr>
<td>Merge</td>
<td>0.09</td>
<td>0.16</td>
<td>0.77</td>
<td>5.0</td>
<td>44.3</td>
<td>550.0</td>
<td>1.7</td>
</tr>
<tr>
<td>Merge/10</td>
<td>0.09</td>
<td>0.21</td>
<td>0.61</td>
<td>0.5</td>
<td>135.1</td>
<td>1400.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Merge/O</td>
<td>0.09</td>
<td>0.22</td>
<td>0.63</td>
<td>4.6</td>
<td>65.0</td>
<td>990.0</td>
<td>2.7</td>
</tr>
<tr>
<td>Heaps</td>
<td>0.08</td>
<td>0.24</td>
<td>0.36</td>
<td>4.0</td>
<td>24.0</td>
<td>1400.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Radix/A</td>
<td>0.08</td>
<td>0.24</td>
<td>0.36</td>
<td>4.0</td>
<td>24.0</td>
<td>1400.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Sorting Lower Bound

We would like to know a lower bound for all possible sorting algorithms.

Sorting is \( \Theta(n \log n) \) (average, worst cases) because we know of algorithms with this upper bound.

Sorting I/O takes \( \Omega(n) \) time.

We will now prove \( \Omega(n \log n) \) lower bound for sorting.

Decision Trees
Lower Bound Proof

- There are $n!$ permutations.
- A sorting algorithm can be viewed as determining which permutation has been input.
- Each leaf node of the decision tree corresponds to one permutation.
- A tree with $n$ nodes has $\Omega(\log n)$ levels, so the tree with $n!$ leaves has $\Omega(\log n!) = \Omega(n \log n)$ levels.

Which node in the decision tree corresponds to the worst case?

Primary vs. Secondary Storage

Primary storage: Main memory (RAM)
Secondary Storage: Peripheral devices
- Disk drives
- Tape drives

Comparisons

<table>
<thead>
<tr>
<th>Medium</th>
<th>Early 1996</th>
<th>Mid 1997</th>
<th>Early 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>$45.00</td>
<td>7.00</td>
<td>1.50</td>
</tr>
<tr>
<td>Disk</td>
<td>0.25</td>
<td>0.10</td>
<td>0.01</td>
</tr>
<tr>
<td>Floppy</td>
<td>0.50</td>
<td>0.36</td>
<td>0.25</td>
</tr>
<tr>
<td>Tape</td>
<td>0.03</td>
<td>0.01</td>
<td>0.001</td>
</tr>
</tbody>
</table>

RAM is usually volatile.
RAM is about 1/4 million times faster than disk.

Golden Rule of File Processing

Minimize the number of disk accesses!
1. Arrange information so that you get what you want with few disk accesses.
2. Arrange information to minimize future disk accesses.
An organization for data on disk is often called a file structure.
Disk-based space/time tradeoff: Compress information to save processing time by reducing disk accesses.

Disk Drives

A sector is the basic unit of I/O.
Interleaving factor: Physical distance between logically adjacent sectors on a track.
**Terms**

**Locality of Reference**: When record is read from disk, next request is likely to come from near the same place in the file.

**Cluster**: Smallest unit of file allocation, usually several sectors.

**Extent**: A group of physically contiguous clusters.

**Internal fragmentation**: Wasted space within sector if record size does not match sector size; wasted space within cluster if file size is not a multiple of cluster size.

**Seek Time**

**Seek time**: Time for I/O head to reach desired track. Largely determined by distance between I/O head and desired track.

**Track-to-track time**: Minimum time to move from one track to an adjacent track.

**Average Seek time**: Average time to reach a track for random access.

**Other Factors**

**Rotational Delay or Latency**: Time for data to rotate under I/O head.
- One half of a rotation on average.
- At 7200 rpm, this is 8.3/2 = 4.2ms.

**Transfer time**: Time for data to move under the I/O head.
- At 7200 rpm: Number of sectors read/Number of sectors per track * 8.3ms.

**Disk Spec Example**

- 16.8 GB disk on 10 platters = 1.68GB/platter
- 13,085 tracks/platter
- 256 sectors/track
- 512 bytes/sector
- Track-to-track seek time: 2.2 ms
- Average seek time: 9.5ms
- 4KB clusters, 32 clusters/track.
- Interleaving factor of 3.
- 5400RPM

**Disk Access Cost Example (1)**

Read a 1MB file divided into 2048 records of 512 bytes (1 sector) each.

Assume all records are on 8 contiguous tracks.

First track: 9.5 + 11.1/2 + 3 x 11.1 = 48.4 ms

Remaining 7 tracks: 2.2 + 11.1/2 + 3 x 11.1 = 41.1 ms.

Total: 48.4 + 7 * 41.1 = 335.7ms

**Disk Access Cost Example (2)**

Read a 1MB file divided into 2048 records of 512 bytes (1 sector) each.

Assume all file clusters are randomly spread across the disk.

256 clusters. Cluster read time is (3 x 8)/256 of a rotation for about 1 ms.

256(9.5 + 11.1/2 + (3 x 8)/256) is about 3877 ms. or nearly 4 seconds.
How Much to Read?

Read time for one track:
9.5 + 11.1/2 + 3 \times 11.1 = 48.4\text{ms}.

Read time for one sector:
9.5 + 11.1/2 + (1/256)11.1 = 15.1\text{ms}.

Read time for one byte:
9.5 + 11.1/2 = 15.05\text{ms}.

Nearly all disk drives read/write one sector at every I/O access.
– Also referred to as a page.

Buffers

The information in a sector is stored in a buffer or cache.

If the next I/O access is to the same buffer, then no need to go to disk.

There are usually one or more input buffers and one or more output buffers.

Buffers

A series of buffers used by an application to cache disk data is called a buffer pool.

Virtual memory uses a buffer pool to imitate greater RAM memory by actually storing information on disk and “swapping” between disk and RAM.

Organizing Buffer Pools

Which buffer should be replaced when new data must be read?

First-in, First-out: Use the first one on the queue.

Least Frequently Used (LFU): Count buffer accesses, reuse the least used.

Least Recently used (LRU): Keep buffers on a linked list. When buffer is accessed, bring it to front. Reuse the one at end.

Buffer Pools

class BufferPool { // (1) Message Passing public:
 virtual void insert(void* space, int sz, int pos) = 0;
 virtual void getbytes(void* space, int sz, int pos) = 0;
};

class BufferPool { // (2) Buffer Passing public:
 virtual void* getblock(int block) = 0;
 virtual void dirtyblock(int block) = 0;
 virtual int blocksize() = 0;
};
Design Issues

Disadvantage of message passing:
- Messages are copied and passed back and forth.
Disadvantages of buffer passing:
- The user is given access to system memory (the buffer itself)
- The user must explicitly tell the buffer pool when buffer contents have been modified, so that modified data can be rewritten to disk when the buffer is flushed.
- The pointer might become stale when the buffer pool replaces the contents of a buffer.

Programmer’s View of Files

Logical view of files:
- An a array of bytes.
- A file pointer marks the current position.

Three fundamental operations:
- Read bytes from current position (move file pointer)
- Write bytes to current position (move file pointer)
- Set file pointer to specified byte position.

C++ File Functions

```cpp
#include <fstream.h>
void fstream::open(char* name, openmode mode);
  // Example: ios::in | ios::binary
void fstream::close();
fstream::read(char* ptr, int numbytes);
fstream::write(char* ptr, int numbyyes);
fstream::seekg(int pos);
fstream::seekg(int pos, ios::curr);
fstream::seekp(int pos);
fstream::seekp(int pos, ios::end);
```

External Sorting

Problem: Sorting data sets too large to fit into main memory.
- Assume data are stored on disk drive.

To sort, portions of the data must be brought into main memory, processed, and returned to disk.

An external sort should minimize disk accesses.

Model of External Computation

Secondary memory is divided into equal-sized blocks (512, 1024, etc...)
A basic I/O operation transfers the contents of one disk block to/from main memory.
Under certain circumstances, reading blocks of a file in sequential order is more efficient. (When?)
Primary goal is to minimize I/O operations.
Assume only one disk drive is available.

Key Sorting

Often, records are large, keys are small.
- Ex: Payroll entries keyed on ID number

Approach 1: Read in entire records, sort them, then write them out again.

Approach 2: Read only the key values, store with each key the location on disk of its associated record.

After keys are sorted the records can be read and rewritten in sorted order.
Simple External Mergesort (1)

Quicksort requires random access to the entire set of records.

Better: Modified Mergesort algorithm.
- Process $n$ elements in $\Theta(\log n)$ passes.

A group of sorted records is called a run.

Simple External Mergesort (2)

- Split the file into two files.
- Read in a block from each file.
- Take first record from each block, output them in sorted order.
- Take next record from each block, output them to a second file in sorted order.
- Repeat until finished, alternating between output files. Read new input blocks as needed.
- Repeat steps 2-5, except this time input files have runs of two sorted records that are merged together.
- Each pass through the files provides larger runs.

Simple External Mergesort (3)

Problems with Simple Mergesort

Is each pass through input and output files sequential?
What happens if all work is done on a single disk drive?
How can we reduce the number of Mergesort passes?

In general, external sorting consists of two phases:
- Break the files into initial runs
- Merge the runs together into a single run.

Breaking a File into Runs

General approach:
- Read as much of the file into memory as possible.
- Perform an in-memory sort.
- Output this group of records as a single run.

Replacement Selection (1)

- Break available memory into an array for the heap, an input buffer, and an output buffer.
- Fill the array from disk.
- Make a min-heap.
- Send the smallest value (root) to the output buffer.
Replacement Selection (2)

- If the next key in the file is greater than the last value output, then
  - Replace the root with this key
- Else
  - Replace the root with the last key in the array

Add the next record in the file to a new heap (actually, stick it at the end of the array).

Snowplow Analogy (1)

Imagine a snowplow moving around a circular track on which snow falls at a steady rate.

At any instant, there is a certain amount of snow $S$ on the track. Some falling snow comes in front of the plow, some behind.

During the next revolution of the plow, all of this is removed, plus 1/2 of what falls during that revolution.

Thus, the plow removes $2S$ amount of snow.

Problems with Simple Merge

Simple mergesort: Place runs into two files.
  - Merge the first two runs to output file, then next two runs, etc.

Repeat process until only one run remains.
  - How many passes for $r$ initial runs?

Is there benefit from sequential reading?
Is working memory well used?
Need a way to reduce the number of passes.

Multiway Merge (1)

With replacement selection, each initial run is several blocks long.
Assume each run is placed in separate file.
Read the first block from each file into memory and perform an $r$-way merge.
When a buffer becomes empty, read a block from the appropriate run file.
Each record is read only once from disk during the merge process.
Multiway Merge (2)
In practice, use only one file and seek to appropriate block.

Limits to Multiway Merge (1)
Assume working memory is $b$ blocks in size.

How many runs can be processed at one time?
The runs are $2b$ blocks long (on average).

How big a file can be merged in one pass?

Limits to Multiway Merge (2)
Larger files will need more passes -- but the run size grows quickly!

This approach trades $(\log b)$ (possibly) sequential passes for a single or very few random (block) access passes.

General Principles
A good external sorting algorithm will seek to do the following:
- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.

Search
Given: Distinct keys $k_1, k_2, \ldots, k_n$ and collection $T$ of $n$ records of the form $(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)$
where $I_j$ is the information associated with key $k_j$ for $1 \leq j \leq n$.

Search Problem: For key value $K$, locate the record $(k_j, I_j)$ in $T$ such that $k_j = K$.
Searching is a systematic method for locating the record(s) with key value $k_j = K$.

Successful vs. Unsuccessful
A successful search is one in which a record with key $k_j = K$ is found.

An unsuccessful search is one in which no record with $k_j = K$ is found (and presumably no such record exists).
Approaches to Search
1. Sequential and list methods (lists, tables, arrays).
2. Direct access by key value (hashing)
3. Tree indexing methods.

Searching Ordered Arrays
- Sequential Search
- Binary Search
- Dictionary Search

Lists Ordered by Frequency
Order lists by (expected) frequency of occurrence.
- Perform sequential search
Cost to access first record: 1
Cost to access second record: 2
Expected search cost:
$$C_n = 1p_1 + 2p_2 + \ldots + np_n.$$ 

Examples(1)
(1) All records have equal frequency.
$$C_n = \sum_{i=1}^{n} \frac{i}{n} = (n+1)/2$$

Examples(2)
(2) Exponential Frequency
$$p_i = \begin{cases} 
1/2^i & \text{if } 1 \leq i \leq n-1 \\
1/2^{n-1} & \text{if } i = n 
\end{cases}$$
$$C_n \approx \sum_{i=1}^{n} \frac{i}{2^i} \approx 2.$$ 

Zipf Distributions
Applications:
- Distribution for frequency of word usage in natural languages.
- Distribution for populations of cities, etc.
$$C_n = \sum_{i=1}^{n} \frac{i}{iH_n} = n/H_n \approx n/\log_n n.$$ 

80/20 rule:
- 80% of accesses are to 20% of the records.
- For distributions following 80/20 rule,
$$C_n \approx 0.1n.$$
Self-Organizing Lists

Self-organizing lists modify the order of records within the list based on the actual pattern of record accesses.

Self-organizing lists use a heuristic for deciding how to reorder the list. These heuristics are similar to the rules for managing buffer pools.

Heuristics

1. Order by actual historical frequency of access. (Similar to LFU buffer pool replacement strategy.)
2. Move-to-Front: When a record is found, move it to the front of the list.
3. Transpose: When a record is found, swap it with the record ahead of it.

Text Compression Example

Application: Text Compression.

Keep a table of words already seen, organized via Move-to-Front heuristic.
• If a word not yet seen, send the word.
• Otherwise, send (current) index in the table.

The car on the left hit the car I left.
The car on 3 left hit 3 5 I 5.
This is similar in spirit to Ziv-Lempel coding.

Searching in Sets

For dense sets (small range, high percentage of elements in set).

Can use logical bit operators.

Example: To find all primes that are odd numbers, compute:
0011010100010100 & 0101010101010101

0 0 1 1 0 1 0 1 0 0 1 0 1 0 0 & 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

Hashing (1)

Hashing: The process of mapping a key value to a position in a table.

A hash function maps key values to positions. It is denoted by \( h \).

A hash table is an array that holds the records. It is denoted by \( HT \).

\( HT \) has \( M \) slots, indexed from 0 to \( M-1 \).

Hashing (2)

For any value \( K \) in the key range and some hash function \( h \), \( h(K) = i \), \( 0 \leq i < M \), such that \( \text{key}(HT[i]) = K \).

Hashing is appropriate only for sets (no duplicates).

Good for both in-memory and disk-based applications.

Answers the question “What record, if any, has key value \( K \)?”
Simple Examples

(1) Store the n records with keys in range 0 to n-1.
   – Store the record with key i in slot i.
   – Use hash function \( h(K) = K \).

(2) More reasonable example:
   – Store about 1000 records with keys in range 0 to 16,383.
   – Impractical to keep a hash table with 16,384 slots.
   – We must devise a hash function to map the key range to a smaller table.

Collisions (1)

Given: hash function \( h \) with keys \( k_1 \) and \( k_2 \).
\( \beta \) is a slot in the hash table.
If \( h(k_1) = \beta = h(k_2) \), then \( k_1 \) and \( k_2 \) have a collision at \( \beta \) under \( h \).

Search for the record with key \( K \):
1. Compute the table location \( h(K) \).
2. Starting with slot \( h(K) \), locate the record containing key \( K \) using (if necessary) a collision resolution policy.

Collisions (2)

Collisions are inevitable in most applications.
   – Example: 23 people are likely to share a birthday.

Hash Functions (1)

A hash function MUST return a value within the hash table range.
To be practical, a hash function SHOULD evenly distribute the records stored among the hash table slots.
Ideally, the hash function should distribute records with equal probability to all hash table slots. In practice, success depends on distribution of actual records stored.

Hash Functions (2)

If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.

If we have knowledge of the incoming distribution, use a distribution-dependent hash function.

Examples (1)

```c
int h(int x) {
    return(x % 16);
}
```
This function is entirely dependent on the lower 4 bits of the key.
Mid-square method: Square the key value, take the middle \( r \) bits from the result for a hash table of \( 2^r \) slots.
Examples (2)
For strings: Sum the ASCII values of the letters and take results modulo $M$.

```c
int h(char* x) {
    int i, sum;
    for (sum=0, i=0; x[i] != '\0'; i++)
        sum += (int) x[i];
    return(sum % M);
}
```

This is only good if the sum is large compared to $M$.

Examples (3)

```c
int ELFhash(char* key) {
    unsigned long h = 0;
    while(*key) {
        h = (h << 4) + *key++;
        unsigned long g = h & 0xF0000000L;
        if (g) h ^= g >> 24;
        h &= ~g;
    }
    return h % M;
}
```

Open Hashing
What to do when collisions occur?
**Open hashing** treats each hash table slot as a bin.

Bucket Hashing
Divide the hash table slots into buckets.
- Example: 8 slots/bucket.

Closed Hashing
Closed hashing stores all records directly in the hash table.
Each record $i$ has a home position $h(k_i)$.
If another record occupies $i$’s home position, then another slot must be found to store $i$.

Collision Resolution
During insertion, the goal of collision resolution is to find a free slot in the table.
Probe sequence: The series of slots visited during insert/search by following a collision resolution policy.
Let $\beta_0 = h(K)$. Let $(\beta_0, \beta_1, \ldots)$ be the series of slots making up the probe sequence.
**Insertion**

```cpp
// Insert e into hash table HT
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const Elem& e) {
    int home; // Home position for e
    int pos = home = h(getkey(e)); // Init
    for (int i = 1;
         !EEComp::eq(EMPTY, HT[pos]); i++) {
        pos = (home + p(getkey(e), i)) % M; // Next
        if (EEComp::eq(e, HT[pos])) // Found it
            return false; // Duplicate
    }
    HT[pos] = e; // Insert e
    return true;
}
```

**Search**

```cpp
// Search for the record with Key K
bool hashdict<Key, Elem, KEComp, EEComp>::hashSearch(const Key& K, Elem& e) const {
    int home; // Home position for K
    int pos = home = h(K); // Initial position
    for (int i = 1; !KEComp::eq(K, HT[pos]) &&
         !EEComp::eq(EMPTY, HT[pos]); i++) {
        pos = (home + p(K, i)) % M; // Next
        if (KEComp::eq(K, HT[pos])) // Found it
            e = HT[pos];
            return true;
    } else return false; // K not in hash table
}
```

**Probe Function**

Look carefully at the probe function \( p() \).

\[
\text{pos} = (\text{home} + p(\text{getkey}(e), i)) \mod M;
\]

Each time \( p() \) is called, it generates a value to be added to the home position to generate the new slot to be examined.

\( p() \) is a function both of the element’s key value, and of the number of steps taken along the probe sequence.

- Not all probe functions use both parameters.

**Linear Probing**

Use the following probe function:

\[
p(K, i) = i;
\]

Linear probing simply goes to the next slot in the table.
- Past bottom, wrap around to the top.

To avoid infinite loop, one slot in the table must always be empty.

**Linear Probing Example**

<table>
<thead>
<tr>
<th>Primary Clustering:</th>
<th>0 1001 1002</th>
<th>6 1981 1982</th>
</tr>
</thead>
<tbody>
<tr>
<td>Records tend to cluster in the table under linear probing since the probabilities for which slot to use next are not the same for all slots.</td>
<td>1 9037 19037</td>
<td>2 3016 193016</td>
</tr>
<tr>
<td>3 2 1</td>
<td>3 3 4</td>
<td>4 4 4</td>
</tr>
<tr>
<td>5 5 6</td>
<td>6 6 6</td>
<td></td>
</tr>
<tr>
<td>7 9874 7 9874</td>
<td>8 2009 8 2009</td>
<td></td>
</tr>
<tr>
<td>9 9875 9 9875</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 10</td>
<td>10 10</td>
<td></td>
</tr>
</tbody>
</table>

**Improved Linear Probing**

Instead of going to the next slot, skip by some constant \( c \).
- Warning: Pick \( M \) and \( c \) carefully.

The probe sequence SHOULD cycle through all slots of the table.
- Pick \( c \) to be relatively prime to \( M \).

There is still some clustering.
- Ex: \( c = 2, h(k_1) = 3; h(k_2) = 5 \).
- Probe sequences for \( k_1 \) and \( k_2 \) are linked together.
Pseudo-Random Probing(1)

The ideal probe function would select the next slot on the probe sequence at random.

An actual probe function cannot operate randomly. (Why?)

Pseudo-Random Probing(2)

• Select a (random) permutation of the numbers from 1 to M-1: r₁, r₂, ..., rₘ.
• All insertions and searches use the same permutation.

Example: Hash table size of M = 101
- r₁=2, r₂=5, r₃=32.
- h(k₁)=30, h(k₂)=28.
- Probe sequence for k₁:
- Probe sequence for k₂:

Quadratic Probing

Set the i'th value in the probe sequence as

\[ h(K, i) = i^2; \]

Example: M=101
- h(k₁)=30, h(k₂) = 29.
- Probe sequence for k₁ is:
- Probe sequence for k₂ is:

Secondary Clustering

Pseudo-random probing eliminates primary clustering.

If two keys hash to the same slot, they follow the same probe sequence. This is called secondary clustering.

To avoid secondary clustering, need probe sequence to be a function of the original key value, not just the home position.

Double Hashing

\[ p(K, i) = i \times h₂(K) \]

Be sure that all probe sequence constants (h₂(K)) are relatively prime to M.
- This will be true if M is prime, or if M=2ᵐ and the constants are odd.

Example: Hash table of size M=101
- h(k₁)=30, h(k₂)=28, h(k₃)=30.
- h₂(k₁)=2, h₂(k₂)=5, h₂(k₃)=5.
- Probe sequence for k₁ is:
- Probe sequence for k₂ is:
- Probe sequence for k₃ is:

Analysis of Closed Hashing

The load factor is \( \alpha = N/M \) where \( N \) is the number of records currently in the table.
Deletion
Deleting a record must not hinder later searches.

We do not want to make positions in the hash table unusable because of deletion.

Tombstones (1)
Both of these problems can be resolved by placing a special mark in place of the deleted record, called a tombstone.

A tombstone will not stop a search, but that slot can be used for future insertions.

Tombstones (2)
Unfortunately, tombstones add to the average path length.

Solutions:
1. Local reorganizations to try to shorten the average path length.
2. Periodically rehash the table (by order of most frequently accessed record).

Indexing
Goals:
- Store large files
- Support multiple search keys
- Support efficient insert, delete, and range queries

Terms (1)
Entry sequenced file: Order records by time of insertion.
  - Search with sequential search

Index file: Organized, stores pointers to actual records.
  - Could be organized with a tree or other data structure.

Terms (2)
Primary Key: A unique identifier for records. May be inconvenient for search.

Secondary Key: An alternate search key, often not unique for each record. Often used for search key.
Linear Indexing

Linear index: Index file organized as a simple sequence of key/record pointer pairs with key values are in sorted order.

Linear indexing is good for searching variable-length records.

Tree Indexing (1)

Linear index is poor for insertion/deletion.

Tree index can efficiently support all desired operations:
- Insert/delete
- Multiple search keys (multiple indices)
- Key range search

2-3 Tree (1)

A 2-3 Tree has the following properties:
1. A node contains one or two keys
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
3. All leaves are at the same level in the tree, so the tree is always height balanced.

The 2-3 Tree has a search tree property analogous to the BST.

Linear Indexing (2)

If the index is too large to fit in main memory, a second-level index might be used.

Tree Indexing (2)

Difficulties when storing tree index on disk:
- Tree must be balanced.
- Each path from root to leaf should cover few disk pages.

2-3 Tree (2)

The advantage of the 2-3 Tree over the BST is that it can be updated at low cost.
2-3 Tree Insertion (1)

2-3 Tree Insertion (2)

2-3 Tree Insertion (3)

The B-Tree is an extension of the 2-3 Tree.

B-Trees (1)

The B-Tree is now the standard file organization for applications requiring insertion, deletion, and key range searches.

B-Trees (2)

1. B-Trees are always balanced.
2. B-Trees keep similar-valued records together on a disk page, which takes advantage of locality of reference.
3. B-Trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

B-Tree Definition

A B-Tree of order $m$ has these properties:
- The root is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and $m$ children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

A B-Tree node is usually selected to match the size of a disk block.
- A B-Tree node could have hundreds of children.
**B-Tree Search (1)**

Search in a B-Tree is a generalization of search in a 2-3 Tree.

1. Do binary search on keys in current node. If search key is found, then return record. If current node is a leaf node and key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.

---

**B+-Trees**

The most commonly implemented form of the B-Tree is the B+-Tree.

Internal nodes of the B+-Tree do not store record -- only key values to guild the search.

Leaf nodes store records or pointers to records.

A leaf node may store more or less records than an internal node stores keys.

---

**B+-Tree Example**

---

**B+-Tree Insertion**

---

**B+-Tree Deletion (1)**

---

**B+-Tree Deletion (2)**
B*-Trees nodes are always at least half full.

The B*-Tree splits two pages for three, and combines three pages into two. In this way, nodes are always 2/3 full.

Asymptotic cost of search, insertion, and deletion of nodes from B-Trees is $\Theta(\log n)$.

– Base of the log is the (average) branching factor of the tree.

Example: Consider a B+-Tree of order 100 with leaf nodes containing 100 records.
1 level B+-tree:
2 level B+-tree:
3 level B+-tree:
4 level B+-tree:

Ways to reduce the number of disk fetches:
– Keep the upper levels in memory.
– Manage B+-Tree pages with a buffer pool.

Graph Applications

• Modeling connectivity in computer networks
• Representing maps
• Modeling flow capacities in networks
• Finding paths from start to goal (AI)
• Modeling transitions in algorithms
• Ordering tasks
• Modeling relationships (families, organizations)

Graphs

A graph $G = (V, E)$ consists of a set of vertices $V$, and a set of edges $E$, such that each edge in $E$ is a connection between a pair of vertices in $V$.

The number of vertices is written $|V|$, and the number edges is written $|E|$.
Paths and Cycles

Path: A sequence of vertices $v_1, v_2, ..., v_n$ of length $n-1$ with an edge from $v_i$ to $v_{i+1}$ for $1 \leq i < n$.

A path is simple if all vertices on the path are distinct.

A cycle is a path of length 3 or more that connects $v_i$ to itself.

A cycle is simple if the path is simple, except the first and last vertices are the same.

Connected Components

An undirected graph is connected if there is at least one path from any vertex to any other.

The maximum connected subgraphs of an undirected graph are called connected components.

Directed Representation

Undirected Representation

Representation Costs

Adjacency Matrix:

Adjacency List:

Graph ADT

class Graph { // Graph abstract class
public:
    virtual int n() = 0; // # of vertices
    virtual int e() = 0; // # of edges
    // Return index of first, next neighbor
    virtual int first(int) = 0;
    virtual int next(int, int) = 0;
    // Store new edge
    virtual void setEdge(int, int, int) = 0;
    // Delete edge defined by two vertices
    virtual void delEdge(int, int) = 0;
    // Weight of edge connecting two vertices
    virtual int weight(int, int) = 0;
    virtual int getMark(int) = 0;
    virtual void setMark(int, int) = 0;
};
Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

The application may require that vertices be visited in some special order based on graph topology.

Examples:
- Artificial Intelligence Search
- Shortest paths problems

Depth First Search (1)

```cpp
// Depth first search
void DFS(Graph* G, int v) {  
    PreVisit(G, v); // Take action  
    G->setMark(v, VISITED);  
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))  
        if (G->getMark(w) == UNVISITED)  
            DFS(G, w);  
    PostVisit(G, v); // Take action  
}
```

Depth First Search (2)

Cost: \(\Theta(|V| + |E|)\).

Breadth First Search (1)

Like DFS, but replace stack with a queue.
- Visit vertex's neighbors before continuing deeper in the tree.

Breadth First Search (2)

```cpp
void BFS(Graph* G, int start, Queue<int>*Q) {  
    int v, w;  
    Q->enqueue(start); // Initialize Q  
    G->setMark(start, VISITED);  
    while (Q->length() != 0) { // Process Q  
        v = Q->dequeue(); // Take action  
        PreVisit(G, v);  
        for (w=G->first(v); w<G->n(); w = G->next(v,w))  
            if (G->getMark(w) == UNVISITED)  
                G->setMark(w, VISITED);  
                Q->enqueue(w);  
    }
}
```
Problem: Given a set of jobs, courses, etc., with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

void topsort(Graph* G) { // Topological sort
    int i;
    for (i=0; i<G->n(); i++) // Initialize
        G->setMark(i, UNVISITED);
    for (i=0; i<G->n(); i++) // Do vertices
        if (G->getMark(i) == UNVISITED)
            tophelp(G, i); // Call helper
}

void tophelp(Graph* G, int v) { // Process v
    G->setMark(v, VISITED);
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))
        if (G->getMark(w) == UNVISITED)
            tophelp(G, w);
    printout(v); // PostVisit for Vertex v
}

void topsort(Graph* G, Queue<int>* Q) { // Topological sort
    int Count[G->n()];
    int v, w;
    for (v=0; v<G->n(); v++) Count[v] = 0;
    for (v=0; v<G->n(); v++) // Process edges
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            Count[w]++; // Add to v2's count
    for (v=0; v<G->n(); v++) // Initialize Q
        if (Count[v] == 0) // No prereqs
            Q->enqueue(v);
    while (Q->length() != 0) {
        v = Q->dequeue(); // PreVisit for V
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (Count[w] == 0) // One less prereq
                Q->enqueue(w);
    }
}

Input: A graph with weights or costs associated with each edge.
Output: The list of edges forming the shortest path.
Sample problems:
- Find shortest path between two named vertices
- Find shortest path from S to all other vertices
- Find shortest path between all pairs of vertices
Will actually calculate only distances.
Shortest Paths Definitions

\[ d(A, B) \] is the shortest distance from vertex \( A \) to \( B \).

\[ w(A, B) \] is the weight of the edge connecting \( A \) to \( B \).

– If there is no such edge, then \( w(A, B) = \infty \).

Single-Source Shortest Paths

Given start vertex \( s \), find the shortest path from \( s \) to all other vertices.

Try 1: Visit vertices in some order, compute shortest paths for all vertices seen so far, then add shortest path to next vertex \( x \).

Problem: Shortest path to a vertex already processed might go through \( x \).

Solution: Process vertices in order of distance from \( s \).

Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>Process A</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>20</td>
<td>\infty</td>
</tr>
<tr>
<td>Process C</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Process B</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process D</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process E</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
</tbody>
</table>

Dijkstra’s Implementation

```c
// Compute shortest path distances from s, return them in D
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;
  for (i=0; i<G->n(); i++) { // Do vertices
    v = minVertex(G, D);
    if (D[v] == INFINITY) return;
    G->setMark(v, VISITED);
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      if (D[w] > (D[v] + G->weight(v, w)))
        D[w] = D[v] + G->weight(v, w);
  }
}
```

Implementing \( \text{minVertex} \)

Issue: How to determine the next-closest vertex? (i.e., implement \( \text{minVertex} \))

Approach 1: Scan through the table of current distances.

– Cost: \( \Theta(|V|^2 + |E|) = \Theta(|V|^2) \).

Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.

– Cost: \( \Theta((|V| + |E|)\log|V|) \)

Approach 1

```c
// Find min cost vertex
int minVertex(Graph* G, int* D) {
  int i, v;
  // Set v to an unvisited vertex
  for (i=G->n(); i>G->n(); i++)
    if (G->getMark(i) == UNVISITED)
      { v = i; break; }
  // Now find smallest D value
  for (i++; i<G->n(); i++)
    if ((G->getMark(i) == UNVISITED) && (D[i] < D[v]))
      { v = i; }
  return v;
}
```
Approach 2

void Dijkstra(Graph* G, int* D, int s) {
    int i, v, w; // v is current vertex
    Dijkstra E[G->e()]; // Heap array
    temp.distance = 0; temp.vertex = s;
    E[0] = temp; // Initialize heap array
    minheap<DijkElem, DDComp> H(E, 1, G->e());
    for (i=0; i<G->n(); i++) {// Get distances
        do { if(!H.removemin(temp)) return;
            v = temp.vertex;
        } while (G->getMark(v) == VISITED);
        G->setMark(v, VISITED);
        if (D[v] == INFINITY) return;
        for(w=G->first(v); w<G->n(); w=G->next(v,w))
            if (D[w] > (D[v] + G->weight(v, w))) {
                D[w] = D[v] + G->weight(v, w);
                temp.distance = D[w]; temp.vertex = w;
                H.insert(temp); // Insert in heap
            } }
}

All-Pairs Shortest Paths

For every vertex $u, v \in V$, calculate $d(u, v)$.
Could run Dijkstra’s Algorithm $|V|$ times.
Better is Floyd’s Algorithm.
Define a $k$-path from $u$ to $v$ to be any path whose intermediate vertices all have indices less than $k$.

Floyd’s Algorithm

//Floyd's all-pairs shortest paths algorithm
void Floyd(Graph* G) {
    int D[G->n()][G->n()]; // Store distances
    for (int i=0; i<G->n(); i++) // Initialize
        for (int j=0; j<G->n(); j++)
            D[i][j] = G->weight(i, j);
    // Compute all k paths
    for (int k=0; k<G->n(); k++)
        for (int i=0; i<G->n(); i++)
            for (int j=0; j<G->n(); j++)
                if (D[i][j] > (D[i][k] + D[k][j]))
                    D[i][j] = D[i][k] + D[k][j];
}

Minimal Cost Spanning Trees

Minimal Cost Spanning Tree (MST)
Problem:
Input: An undirected, connected graph $G$.
Output: The subgraph of $G$ that
1) has minimum total cost as measured by summing the values of all the edges in the subset, and
2) keeps the vertices connected.

Minimal Cost Spanning Tree Algorithm

void Prim(Graph* G, int* D, int s) {
    int V[G->n()]; // Who's closest
    int i, w;
    for (i=0; i<G->n(); i++) // Do vertices
        int v = minVertex(G, D, G->getMark(v, VISITED));
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return;
    for (w=G->first(v); w<G->n(); w = G->next(v, w))
        if (D[w] > G->weight(v, w)) {
            D[w] = G->weight(v, w); // Update dist
            V[w] = v; // Update who it came from
        }
}
Alternate Implementation
As with Dijkstra’s algorithm, the key issue is determining which vertex is next closest.
As with Dijkstra’s algorithm, the alternative is to use a priority queue.
Running times for the two implementations are identical to the corresponding Dijkstra’s algorithm implementations.

Kruskal’s MST Algorithm (1)
Initially, each vertex is in its own MST.
Merge two MST’s that have the shortest edge between them.
– Use a priority queue to order the unprocessed edges. Grab next one at each step.
How to tell if an edge connects two vertices already in the same MST?
– Use the UNION/FIND algorithm with parent-pointer representation.

Kruskal’s MST Algorithm (2)
Cost is dominated by the time to remove edges from the heap.
– Can stop processing edges once all vertices are in the same MST
Total cost: \( \Theta(|V| + |E| \log |E|) \).