Introduction to Data Structures

For : COP 3330.

Object oriented Programming (Using C++)

http://www.compgeom.com/~piyush/teach/3330

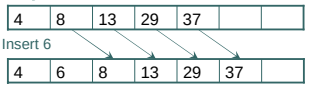
Piyush Kumar

Sorted Arrays As Lists

- Arrays are used to store a list of values
- Arrays are contained in contiguous memory
  - Recall – inserting a new element in the middle of an array requires later elements to be "shifted". For large lists of values, this is inefficient.

```
void insertSorted(int value, int &length, int list[])
{
  int i = length - 1;
  while (list[i] > value)
  {
    list[i + 1] = list[i];
    i--;
  }
  list[i + 1] = value;
  length++;
}
```

Shifting array elements that come after the element being inserted



Sorted Arrays

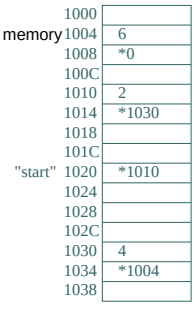
- Due to the need to "shift" elements, sorted arrays are:
  - Inefficient when inserting into the middle or front
  - Inefficient when deleting from the middle or front
  - Efficient for searching
- Since inserting and deleting are common operations, we need to find a data structure which allows more efficiency
  - Contiguous memory will not work – will always require a shift
  - "Random" placement requires "random" memory locations
  - Dynamic allocation provides "random" locations, and means that the list can grow as much as necessary
  - The maximum size need not be known – ever
    - This is not true for arrays, even dynamically allocated arrays

Intro To Linked Lists

- A linked list is a data structure which allows efficient insertion and deletion.
- Consists of "nodes". Each node contains:
  - A value - the data being stored in the list
  - A pointer to another (the next) node
- By carefully keeping pointers accurate, you can start at the first node, and follow pointers through entire list.
- Graphically, linked list nodes are represented as follows:

Linked List Info

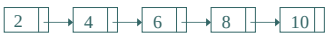
- Each node is dynamically allocated, so memory placement is "random"



The above linked list may be stored in memory as shown to the right.

Deletion From Linked Lists

Given the initial linked list:



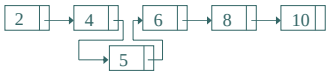
Delete node with value 4

Resulting in

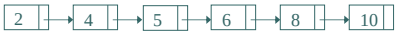
Insertion Into Linked Lists

- Given the initial linked list:

Insert node with value 5



Resulting in



Linked List Nodes, Example

Use a class to group the value and the pointer

```
class ListNodeClass
{
  public:                 // Only for illustration: Bad design
    int val;              //Will have a list of ints
    ListNodeClass *next;  //Point to the next node
};

int main()
{
  ListNodeClass *head = 0;  //Essentially, this declares a
                            //list, since it will point to the
                            //first node of a list.  Initially,
                            //list is empty (null pointer)

  head = new ListNodeClass; //Note no [] - not
                            //declaring an array - just
                            //one single node
...
```

Printing a List (Visiting Each Node)

```
void printList(ListNodeClass *head)
{
  ListNodeClass *temp = head;

  if (temp == 0)
  {
    cout << "List is Empty!" << endl;
  }
  else
  {
    while (temp != 0)
    {
      cout << temp->val << " ";
      temp = temp->next;
    }
    cout << endl;
  }
}
```

Try To Insert To Front Of List

```
bool insertAtHead1(ListNodeClass *head,
                   int newVal)
{
  bool status = true;
  ListNodeClass *temp;

  temp = new ListNodeClass;
  if (temp == 0)
  {
    cout << "Unable to alloc node"
         << endl;
    status = false;
  }
  else
  {
    temp->val = newVal;

    if (head == 0)
    {
      temp->next = 0;
      head = temp;
    }
    else
    {
      temp->next = head;
      head = temp;
    }
  }
  return (status);
}
int main(void)
{
  ListNodeClass *head1 = 0;
  printList(head1);
  insertAtHead1(head1, 5);
  insertAtHead1(head1, 8);
  insertAtHead1(head1, 17);
  printList(head1);
  return (0);
}
```

List is Empty!
List is Empty!

Corrected Insert To Front Of List

```
bool insertAtHead(ListNodeClass **head,
                  int newVal)
{
  bool status = true;
  ListNodeClass *temp;

  temp = new ListNodeClass;
  if (temp == 0)
  {
    cout << "Unable to alloc node"
         << endl;
    status = false;
  }
  else
  {
    temp->val = newVal;

    if (*head == 0)
    {
      temp->next = 0;
      *head = temp;
    }
    else
    {
      temp->next = *head;
      *head = temp;
    }
  }
  return (status);
}

int main(void)
{
  ListNodeClass *head = 0;
  printList(head);
  insertAtHead(&head, 5);
  insertAtHead(&head, 8);
  insertAtHead(&head, 17);
  printList(head);
  return (0);
}
```

List is Empty!
17 8 5

Reference to pointer example

```
// Reference to pointer example
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
        int* i = 0;
        cout << "i = " << i << endl;
        increment(i);
        cout << "i = " << i << endl;
}
```

0
4

Deleting From Front Of List

```
bool deleteFromFront(
     ListNodeClass **head)
{
  bool status = true;
  ListNodeClass *temp;

  if (*head == 0)
  {
    cout << "Can't delete from list"
         << endl;
    status = false;
  }
  else
  {
    temp = *head;
    *head = temp->next;
    //Free the memory we dynamically
    //allocated in insert function
    delete temp;
  }
  return (status);
}
```

```
int main(void)
{
  ListNodeClass *head = 0;
  printList(head);
  insertAtHead(&head, 5);
  insertAtHead(&head, 8);
  insertAtHead(&head, 17);
  printList(head);
  deleteFromFront(&head);
  printList(head);
  deleteFromFront(&head);
  deleteFromFront(&head);
  printList(head);
  deleteFromFront(&head);
  return (0);
}
```

```
List is Empty!
17 8 5
8 5
List is Empty!
Cannot delete from list!
```

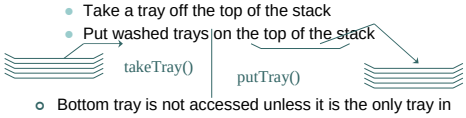
Searching A List

```
bool searchList(ListNodeClass *head, int val)
{
  bool found = false;
  ListNodeClass *temp = head;

  while (temp != 0 && !found)
  {
    if (temp->val == val)
    {
      found = true;
    }
    else
    {
      temp = temp->next;
    }
  }
  return (found);
}
```

The Stack Linked Structure

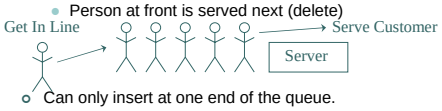
- A stack is another data structure
  - Used to organize data in a certain way
- Think of a stack as a stack of cafeteria trays
  - Take a tray off the top of the stack
  - Put washed trays on the top of the stack



- Bottom tray is not accessed unless it is the only tray in the stack.
- Since only the top of a stack can be accessed, there needs to be only one insert function and one delete function
  - Inserting to a stack is usually called "push"
  - Deleting from a stack is usually called "pop"

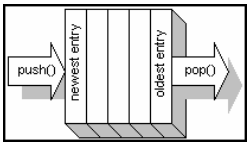
The Queue Linked Structure

- A queue is another data structure.
- Think of a queue as a line of people at a store
  - Get into the line at the back (insert)
  - Person at front is served next (delete)



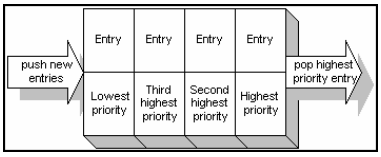
- Can only insert at one end of the queue.
  - Inserting to a queue is usually called "enqueue()"
  - "Get In Line" in above diagram
- Can only remove at the other end of the queue
  - Removing from a queue is usually called "dequeue()"
  - "Serve Customer" in above diagram

Another Queue Pic: FIFO



- First In First out.

Priority Queue Pic: FIFO



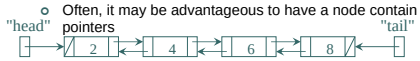
- Highest priority goes out first.
- The **STL** has three container adaptor types: **stack**, queue, and **priority_queue**.

The Priority Queue Linked Structure

- A priority queue works slightly differently than a "normal" queue
- Elements in a priority queue are sorted based on a priority
  - Queue order is not dependent on the order in which elements were inserted, as it was for a normal queue
  - As elements are inserted, they are sorted such that the element with the highest priority is at the beginning of the priority queue
  - When an element is removed from the priority queue, the first element (highest priority) is taken, regardless of when it was inserted
  - Elements of the same priority are maintained in the order which they were inserted
- Using a priority queue in which all elements have the same priority is equivalent to using a "normal" queue

The Doubly-Linked List Structure

- The linked list examples we've seen so far have only one pointer
- Often, it may be advantageous to have a node contain multiple pointers




```
class DoublyLinkedListNodeClass
{
  DoublyLinkedListNodeClass *prev;
  int val;
  DoublyLinkedListNodeClass *next;
};

DoublyLinkedListNodeClass *head;
DoublyLinkedListNodeClass *tail;
```

STL list<>

- A list is a doubly linked list.
- Example

```
list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

http://www.sgi.com/tech/stl/List.html

Note that singly linked lists, which only support forward traversal, are also sometimes useful. If you do not need backward traversal, then slist may be more efficient than list.

Container Adaptors

- Container adaptors take sequence containers as their type arguments, for example:
  - stack < vector < int > > a;

Stack (Last In First Out)

- Use with vector (best/default),
  - deque, or list (bad choice)
- Basic interface:
  - bool empty();
  - size_type size();
  - value_type& top();
  - const value_type& top();
  - void push(const value_type&);
  - **void** pop();

Stack

#include <stack>

Provides: **stack<T, Sequence>**

Container Function	Stack Adapter Function
back()	top()
push_back()	push()
pop_back()	pop()
empty()	empty()
size()	size()

To support this functionality stack expects the underlying container to support push_back(), pop_back(), empty() or size() and back()

Stack Example.

```
// C++ STL Headers
#include <iostream>
#include <vector>
#include <stack>

int main( int argc, char *argv[] )
{
   stack<const char *,  vector<const char *> > s;

//  Push on stack in reverse order
   s.push("order");
   s.push("correct"); // Oh no it isn't !
   s.push("the");
   s.push("in");
   s.push("is");
   s.push("This");

//  Pop off stack which reverses the push() order
   while ( !s.empty() ) {
     cout  << s.top() <<" "; s.pop(); /// Oh yes it is !
   }
   cout << endl;

   return( EXIT_SUCCESS );
}
```


Simpler Stack example

```
int main() {
  stack<int> S;
  S.push(8);
  S.push(7);
  S.push(4);
  assert(S.size() == 3);

  assert(S.top() == 4);
  S.pop();

  assert(S.top() == 7);
  S.pop();

  assert(S.top() == 8);
  S.pop();

  assert(S.empty());
}
```

http://www.sgi.com/tech/stl/stack.html

Queue

- Use with deque (default), or list. (Vector works, but its extremely inefficient)
- Basic interface:
  - bool empty();
  - size_type size();
  - value_type& front();
  - const value_type& front();
  - value_type& back();
  - const value_type& back();
  - void push(const value_type&);
  - **void** pop();

```
#include <queue>
#include <vector>
#include <list>
#include <iostream>

int main( ) { using namespace std;

// Declares queue with default deque base container
queue <char> q1;
// Explicitly declares a queue with deque base container
queue <char, deque<char> > q2;

// These lines don't cause an error, even though they
// declares a queue with a vector base container
queue <int, vector<int> > q3;
q3.push( 10 );

// but the following would cause an error because vector has
// no pop_front member function
// q3.pop( );
// Declares a queue with list base container
queue <int, list<int> > q4;

// The second member function copies elements from a container
list<int> li1; li1.push_back( 1 ); li1.push_back( 2 );
queue <int, list<int> > q5( li1 );
cout << "The element at the front of queue q5 is " << q5.front( ) << "." << endl;
cout << "The element at the back of queue q5 is " << q5.back( ) << "." << endl;
}
```

```
int main() {
  priority_queue<int> Q;
  Q.push(1);
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);

  assert(Q.top() == 8);
  Q.pop();

  assert(Q.top() == 7);
  Q.pop();

  assert(Q.top() == 5);
  Q.pop();

  assert(Q.top() == 4);
  Q.pop();

  assert(Q.top() == 2);
  Q.pop();

  assert(Q.top() == 1);
  Q.pop();

  assert(Q.empty());
}
```

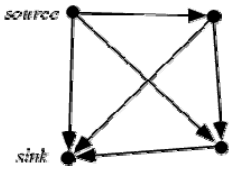
Graphs

An introduction

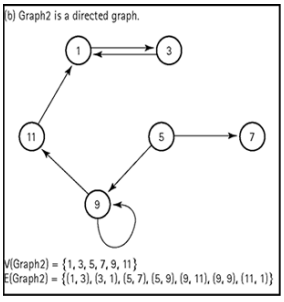
Graphs

- A **graph G** = **(**V,E**)** is composed of:
  - V: set of **vertices**
  - E ⊂ V × V: set of **edges** connecting the **vertices**
- An **edge *e*** = (*u*,*v*) is a __ pair of vertices
  - Directed graphs (ordered pairs)
  - Undirected graphs (unordered pairs)

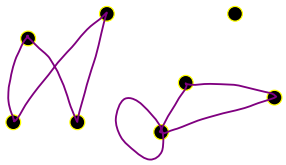
Directed graph



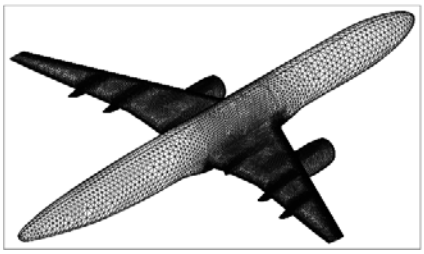
Directed Graph



Undirected GRAPH



Undirected Graph

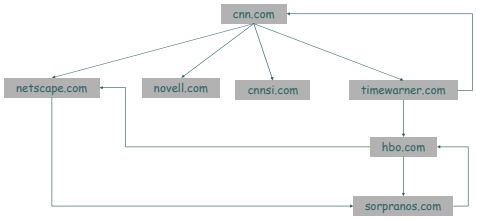


Some More Graph Applications

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

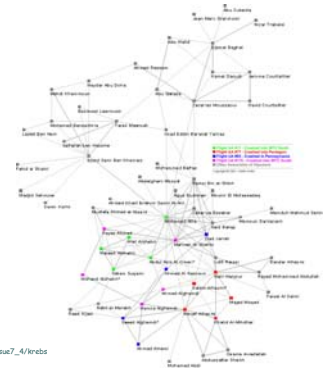
World Wide Web

- Web graph.
  - Node: web page.
  - Edge: hyperlink from one page to another.



9-11 Terrorist Network

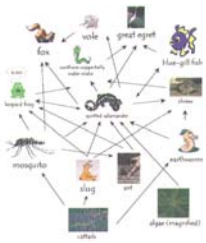
- Social network graph.
  - Node: people.
  - Edge: relationship between two people.



Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

- Food web graph.
  - Node = species.
  - Edge = from prey to p



Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

Terminology

- **a** is adjacent to **b** iff (**a**,**b**) $\in$ **E**.
- *degree*(a) = number of adjacent vertices (Self loop counted twice)
- Self Loop: (a,a)

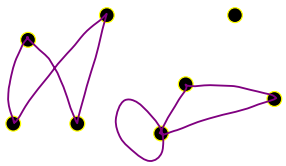
- Parallel edges: E = { ...(a,b), (a,b)...}

Terminology

- A Simple Graph is a graph with no self loops or parallel edges.
- Incidence: v is incident to e if v is an end vertex of e.

Question

- Max Degree node? Min Degree Node? Isolated Nodes? Total sum of degrees over all vertices? Number of edges?



QUESTION

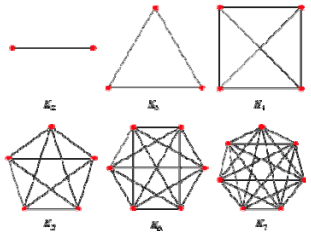
- How many edges are there in a graph with 100 vertices each of degree 4?

Connected graph

- Undirected Graphs: If there is at least one path between every pair of vertices. (otherwise disconnected)

complete graph

- Every pair of graph vertices is connected by an edge.



n(n-1)/2 edges

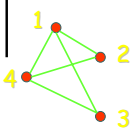
Trees

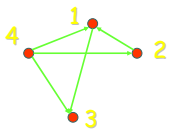
- An undirected graph is a tree if it is connected and does not contain a cycle.
- Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.

representation

- Two ways
  - Adjacency List
    - (as a linked list for each node in the graph to represent the edges)
  - Adjacency Matrix
    - (as a boolean matrix)

Representing Graphs

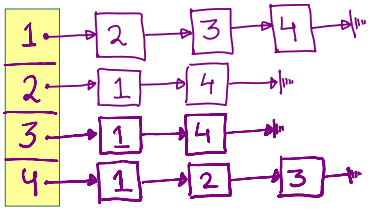




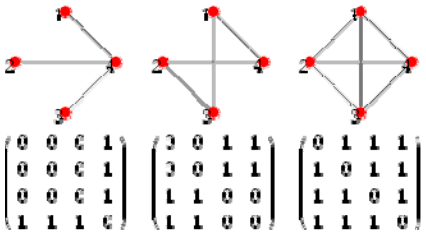
Vertex	Adjacent Vertices
1	2, 3, 4
2	1, 4
3	1, 4
4	1, 2, 3

Initial Vertex	Terminal Vertices
1	3
2	1
3	
4	1, 2, 3

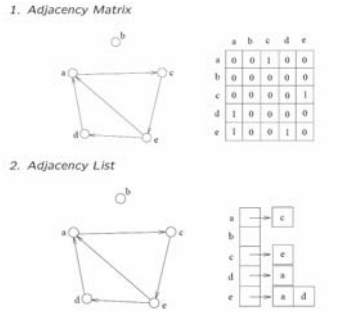
adjacency list



adjacency matrix



Another example



AL Vs AM

- AL : Total space = 4|V| + 8|E| bytes (For undirected graphs its 4|V| + 16|E| bytes)
- AM : |V| * |V| / 8
- Question: What is better for very sparse graphs? (Few number of edges)

AL Vs AM

- Question: How much time does it take to find out if (v_i,v_j) belongs to E?
  - AM ?
  - AL ?

Stable Marriage

Our next problem

The problem

- There are n men and n women
- Each man has a preference list, so does the woman.
- These lists have no ties.

- Devise a system by which each of the n men and n women can end up getting married.

Other Similar problems

- Given a set of colleges and students pair them. (Internship – Company assignments)
- Given airlines and pilots, pair them.
- Given two images, pair the points belonging to the same point in 3D to extract depth from the two images.
- Dorm room assignments.
- Hospital residency assignments**.
- Your first programming assignment…

Stereo Matching

Fact: If one knows the distance between the cameras
And the matching, its almost trivial to recover depth..

Example Preference Lists

Man	1st	2nd	3rd
X	A	B	C
Y	B	A	C
Z	A	B	C

Woman	1st	2nd	3rd
A	Y	X	Z
B	X	Y	Z
C	X	Y	Z

What goes wrong?

Unstable pairs: (X,C) and (B,Y)
They prefer each other to current pairs.

Stable Matching

Man	1st	2nd	3rd
X	A	B	C
Y	B	A	C
Z	A	B	C

Woman	1st	2nd	3rd
A	Y	X	Z
B	X	Y	Z
C	X	Y	Z

No Pairs creating ***instability***.

Another Stable Matching

Man	1st	2nd	3rd
X	A	B	C
Y	B	A	C
Z	A	B	C

Woman	1st	2nd	3rd
A	Y	X	Z
B	X	Y	Z
C	X	Y	Z

Stability is Primary.

- Any reasonable list of criteria must contain the stability criterion.
- A pairing is doomed if it contains a shaky couple.

Main Idea

Idea: Allow the pairs to keep breaking up and reforming until they become stable

Can you argue that the couples will not continue breaking up and reforming forever?

Men Propose (Women dispose)

```
Initialize each person to be free.
while (some man m is free and hasn't proposed to every woman)
        w = first woman on m's list to whom m has not yet proposed
        if  (w is free)
              assign m and w to be engaged
        else if  (w prefers m to her fiancé m')
              assign m and w to be engaged, and m' to be free
        else
        w rejects m
```

Gale-Shapley Algorithm (men propose)