Fast sorting algorithms: mergesort, quicksort, heapsort?
how they work   why/when they're fast

In-order Pre-order Post-order for trees: 10/23 Notes

"In computer science, **tree traversal** refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well."

- Linked lists, one dimensional arrays, are all linear data structures with one standard method of tranversal. Tree structures can be traversed in many ways.
- There are three steps you can do on a tree at any point: 1) perform action on current node 2) traversing to left child node 3) traversing to right child node.
- Due to multiple possible nodes to be visited, nodes must be stored for later.
- Storage can be done by a stack (First in Last out) or a queue (First in First out)
- Depth First: Going down next level first ie. first child, grandchild before second child
- Breadth First: Going horizontally ie. first child, second child, grandchildren.
- Depth First has different versions related to order of transversal (left or right), there is only one version of breadth-first.
- Depth First transversal is done easiest by stack, breadth first is easiest done by a queue.

Types of Depth-first transversal:

Preorder:

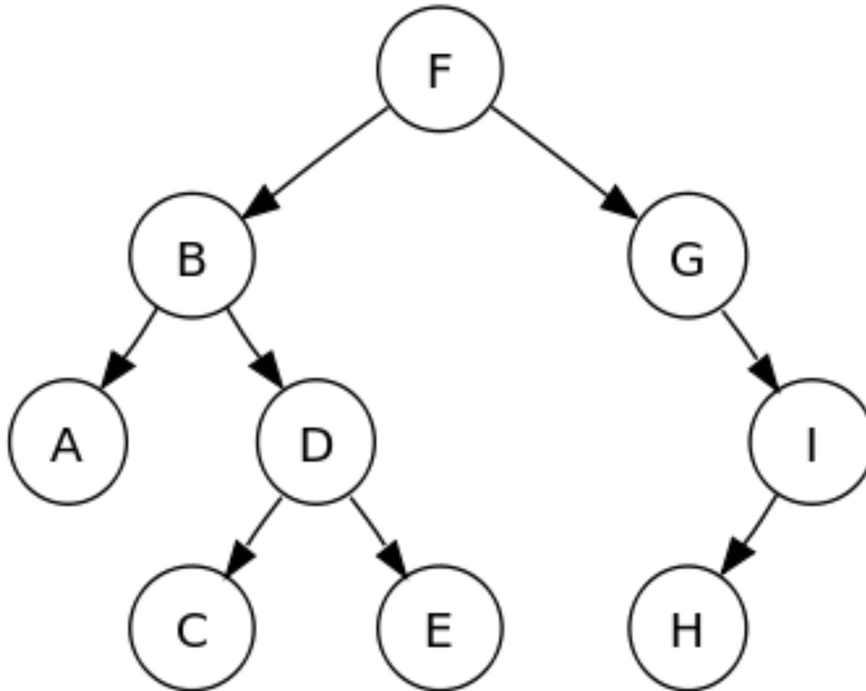
1) Visit the root
2) Traverse the left subtree
3) Traverse the left subtree

Inorder (symmetric):

1) Traverse the left subtree
2) Visit the root
3) Traverse the right subtree

Postorder:

1) Traverse the left subtree
2) Traverse the right subtree
3) Visit the root



Depth-First
Preorder: F, B, A, D, C, E, G, I, H (root,left,right)
Inorder: A, B, C, D, E, F, G, H, I (left,root, right)
Postorder: A, C, E, D, B, H, I, G, F (left,right, root)
Breadth First: F, B, G, A, D, I, C, E, H

In-Order:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct node
{ char * name;
  node * left, * right;
  node(char *);
  node(char *, node *, node *); };

node::node(char * n)
{ name = n;
  left = NULL;
  right = NULL; }       //constructor

node::node(char * n, node * l, node * r)
```

```
{ name = n;
  left = l;
  right = r; }          //constructor

char * char_to_string(char c)  // this is used later
{ char * s = new char[2];
  s[0] = c;
  s[1] = 0;
  return s; }

node * insert(node * t, char * s)
{ if (t == NULL)
    return new node(s);
  if (strcmp(s, t->name) < 0)
    t->left = insert(t->left, s);
  else
    t->right = insert(t->right, s);
  return t; }

void print(node * t)
{ if (t == NULL)
    printf("-");
  else
  { printf("(");
    print(t->left);
    printf("%s", t->name);
    print(t->right);
    printf(")"); } }

void niceprint(node * t, int indent)
{ if (t == NULL)
    return;
  niceprint(t->right, indent+1);
  printf("%*s%s\n", indent*2, "", t->name);
  niceprint(t->left, indent+1); }

char * items[] = { "M", "K", "U", "F", "L", "Q", "Y", "D", "H", "O",
                   "S", "W", "Z", "B", "E", "G", "I", "N", "P", "R",
                   "T", "V", "X", "A", "C", "J" };

const int nitems = sizeof(items)/sizeof(items[0]);

void main()
{ node * root = NULL;
  for (int i = 0; i < nitems; i += 1)
    root = insert(root, items[i]);
  printf("in-order print:\n");
  print(root);
  printf("\nnice print:\n");
  niceprint(root, 0); }

Preorder Print Function:
void print(node * t)
  { if (t == NULL)
```

```
      printf("-");
    else
    { printf("%s", t->name);
      print(t->left);
      print(t->right); } }
```

```
Preorder Reconstruction:
node * readtree()
  { int c = getchar();
    if (c=='-')
      return NULL;
    else
    { node * t = new node(char_to_string(c));
      t->left = readtree();
      t->right = readtree();
      return t; } }
```

```
Post-Order Print:
 void print(node * t)
  { if (t == NULL)
      printf("-");
    else
    { print(t->left);
      print(t->right);
      printf("%s", t->name); } }
```

```
Post-Order Reconstruction:
  node * readtree()
   { stack <node *> s;
     while (true)
      { int c = getchar();

        if (c == '\n')
          break;

        else if (c == '-')
          s.add(NULL);

        else
         { node * right = s.take();
           node * left = s.take();
           node * t = new node(char_to_string(c), left, right);
           s.add(t); } }

    return s.take(); }
```

Graph First Exploration:

Depth First Graph Exploration:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <assert.h>

template <typename T> class stack
{ protected:

    struct entry
    { T value;
      entry * next;
      entry(T, entry *); };

    entry * end;

  public:

    stack();
    bool empty();
    void add(T);
    T take(); };

template <typename T> stack <T> :: entry :: entry(T v, entry * prev)
{ value = v;
  next = prev; }

template <typename T> stack <T> :: stack()
{ end = NULL; }

template <typename T> bool stack <T> :: empty()
{ return end == NULL; }

template <typename T> void stack <T> :: add(T v)
{ end = new entry(v, end); }

template <typename T> T stack <T> :: take()
{ assert(end != NULL);
  T v = end->value;
  entry * old_end = end;
  end = end->next;
  delete old_end;
  return v; }

struct node
{ char * name;
  node * left, * right;
  int time;
```

```
  node(char, node *, node *); };

node::node(char c, node * l, node * r)
{ char * s = new char[2];
  s[0] = c;
  s[1] = 0;
  name = s;
  left = l;
  right = r;
  time = INT_MAX; }

node * wholeworld[128];

void connect(char na, char nb)
{ node * a, * b;

  a = wholeworld[na];
  if (a == NULL)
  { fprintf(stderr, "Node does not exist: '%c'\n", na);
    exit(1); }

  b = wholeworld[nb];
  if (b == NULL)
  { fprintf(stderr, "Node does not exist: '%c'\n", nb);
    exit(1); }

  if (a->left == NULL)
  { a->left = b;
    return; }

  if (a->right == NULL)
  { a->right = b;
    return; }

  fprintf(stderr, "Node '%c', too many connections\n", na);
  exit(1); }

void init()
{ for (char c = 'A'; c <= 'Z'; c += 1)
  { wholeworld[c] = new node(c, NULL, NULL);
    wholeworld[tolower(c)] = wholeworld[c]; }
  connect('A', 'C');
  connect('A', 'B');
  connect('B', 'E');
  connect('B', 'D');
  connect('C', 'N');
  connect('C', 'F');
  connect('D', 'K');
  connect('D', 'J');
  connect('E', 'G');
```

```
  connect('F', 'H');
  connect('F', 'I');
  connect('G', 'N');
  connect('G', 'L');
  connect('H', 'N');
  connect('H', 'M');
  connect('I', 'O');
  connect('I', 'M');
  connect('J', 'K');
  connect('J', 'Q');
  connect('K', 'L');
  connect('L', 'N');
  connect('L', 'Q');
  connect('M', 'O');
  connect('N', 'P');
  connect('O', 'P');
  connect('X', 'C');
  connect('X', 'Z');
  connect('Y', 'Z');
  connect('Z', 'F');
  connect('Z', 'I'); }

void consider(node * n, stack <node *> & s, int when)
{ if (n != NULL && n->time > when)
  { n->time = when;
    s.add(n); } }

void main()
{ stack <node *> s;
  init();
  node * n = wholeworld['A'];
  s.add(n);
  n->time = 0;
  while (! s.empty())
  { node * x = s.take();
    int timenow = x->time;
    printf("%s(%d)  ", x->name, timenow);
    consider(x->left, s, timenow+1);
    consider(x->right, s, timenow+1); }
  printf("\n"); }
```

- Not recursive but driven by a stack, results in a lot of repeated exploration.

Breadth-First Search:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#include <limits.h>
#include <assert.h>

template <typename T> class queue
{ protected:

    struct entry
    { T value;
      entry * next;
      entry(T, entry *); };

    entry * front, * end;

  public:

    queue();
    bool empty();
    void add(T);
    T take(); };

template <typename T> queue <T> :: entry :: entry(T v, entry * prev)
{ value = v;
  next = prev; }

template <typename T> queue <T> :: queue()
{ front = NULL;
  end = NULL; }

template <typename T> bool queue <T> :: empty()
{ return end == NULL; }

template <typename T> void queue <T> :: add(T v)
{ entry * e = new entry(v, NULL);
  if (front == NULL)
    front = e;
  else
    end->next = e;
  end = e; }

template <typename T> T queue <T> :: take()
{ assert(front != NULL);
  T v = front->value;
  entry * old_front = front;
  front = front->next;
  if (front == NULL)
    end = NULL;
  delete old_front;
  return v; }

struct node
```

```
{ char * name;
  node * left, * right;
  int time;
  node(char, node *, node *); };

node::node(char c, node * l, node * r)
{ char * s = new char[2];
  s[0] = c;
  s[1] = 0;
  name = s;
  left = l;
  right = r;
  time = INT_MAX; }

node * wholeworld[128];

void connect(char na, char nb)
{ node * a, * b;

  a = wholeworld[na];
  if (a == NULL)
  { fprintf(stderr, "Node does not exist: '%c'\n", na);
    exit(1); }

  b = wholeworld[nb];
  if (b == NULL)
  { fprintf(stderr, "Node does not exist: '%c'\n", nb);
    exit(1); }

  if (a->left == NULL)
  { a->left = b;
    return; }

  if (a->right == NULL)
  { a->right = b;
    return; }

  fprintf(stderr, "Node '%c', too many connections\n", na);
  exit(1); }

void init()
{ for (char c = 'A'; c <= 'Z'; c += 1)
  { wholeworld[c] = new node(c, NULL, NULL);
    wholeworld[tolower(c)] = wholeworld[c]; }
  connect('A', 'C');
  connect('A', 'B');
  connect('B', 'E');
  connect('B', 'D');
  connect('C', 'N');
  connect('C', 'F');
```

```
  connect('D', 'K');
  connect('D', 'J');
  connect('E', 'G');
  connect('F', 'H');
  connect('F', 'I');
  connect('G', 'N');
  connect('G', 'L');
  connect('H', 'N');
  connect('H', 'M');
  connect('I', 'O');
  connect('I', 'M');
  connect('J', 'K');
  connect('J', 'Q');
  connect('K', 'L');
  connect('L', 'N');
  connect('L', 'Q');
  connect('M', 'O');
  connect('N', 'P');
  connect('O', 'P');
  connect('X', 'C');
  connect('X', 'Z');
  connect('Y', 'Z');
  connect('Z', 'F');
  connect('Z', 'I'); }

void consider(node * n, queue <node *> & s, int when)
{ if (n != NULL && n->time > when)
  { n->time = when;
    s.add(n); } }

void main()
{ queue <node *> s;
  init();
  node * n = wholeworld['A'];
  s.add(n);
  n->time = 0;
  while (! s.empty())
  { node * x = s.take();
    int timenow = x->time;
    printf("%s(%d)  ", x->name, timenow);
    consider(x->left, s, timenow+1);
    consider(x->right, s, timenow+1); }
  printf("\n"); }
```

- Breadth-first Search uses a queue instead of a stack.
- Breadth-first search explores in layers, visiting everything N steps away before anything N+1 steps away.
- It is a much quicker way to find the shortest path.

Abstract Data Types:
A mathematical model for a class of data structures that have similar behavior. It is defined indirectly through the operations that may be performed on it as well as its mathematical constraints on the effects.

Queue:
- Abstract data type where the entities are kept in order and the only operations are addition to the rear terminal and removal from the front terminal position (FIFO)
- Once an element is added, all other elements that were added before have to be removed before the new element can be removed (linear data structure)

Priority Queue:
- Simialar to a normal queue, except each element has a priority associated with it, high priority elements are served before low priority ones.
- If elements have the same priority they are served according to their place in th queue.

Heap:
- Specialized tree-based data structure that satisfies the heap property.
- Heap property: If A is a parent node of B, then key(A) is always ordered with respect to key(B) with the same ordering appling across the heap.
- Keys of parent nodes are either always greater than or equal to those of the chldren and the highest key is in the root node.
- Keys of parent nodes area less than or equal to those of children.

heaps: 11/8, 11/13

Templates? Concrete/abstract?