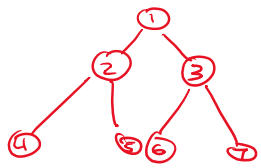
# All forms of DFS



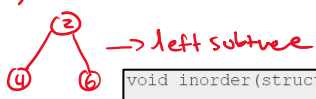

Algorithm: Visit root node, recurse thru left subtree, recurse thru right subtree.

## Binary Tree Traversal
- Pre-order
- In order
- Post order

**Lab Section 13**
4-5:50pm
Sage 3101
TA: Matthew
mentors: Aditya & Kajsa & Tyler H.

## Pre-order (NLR)
- First node is root

(1,2,4,5,3,6,7).





```
void preorder(struct node *root) {
   if (root != NULL) {
      cout<<root->data<<" ";
      preorder(root->left);
      preorder(root->right);
   }
}
```

```
void inorder(struct node *root) {
   if (root != NULL) {
      inorder(root->left);
      cout<<root->data<<" ";
      inorder(root->right);
   }
}
```

## In-Order (L,N,R)
(4,2,5,1,6,3,7)

Algorithm: recurse down left tree, then node, then right

## Post-order (LRN) (4,5,2,6,7,3,1)
Algorithm: At each node first go left, right, then that node

```
void postorder(struct node *root) {
   if (root != NULL) {
      postorder(root->left);
      postorder(root->right);
      cout<<root->data<<" ";
   }
}
```

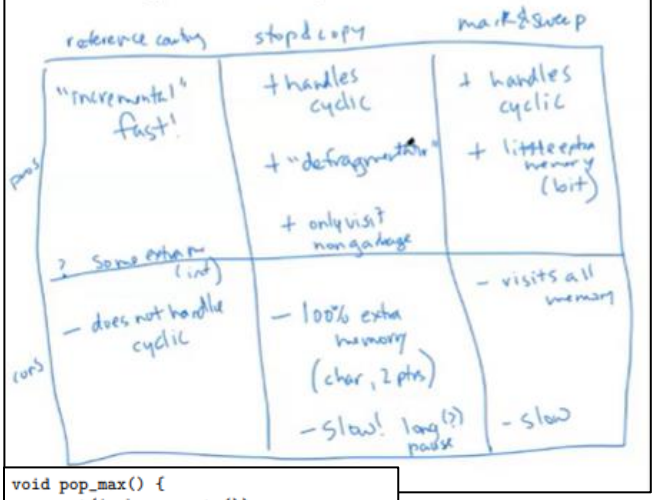
```
void reverse()
{
    // Initialize current, previous a
    Node* current = head;
    Node *prev = NULL, *next = NULL;

    while (current != NULL) {
        // Store next
        next = current->next;
        // Reverse current node's poi
        current->next = prev;
        // Move pointers one position
        prev = current;
        current = next;
    }
    head = prev;
}
```

```
Superhero& operator/=(const string &id);
```
member function

| | |
|---|---|
| default constructor | Office(); |
| destructor | ~Office(); |
| copy constructor | Office(const Office &office); |
| assignment operator | Office& operator=(const Office &office); |

**Solution:**
```
void RecursiveFor(int i,int max){
  if(i<max){
    x();
    RecursiveFor(i+1,max);
  }
}
```

### 27.10 Garbage Collection Comparison



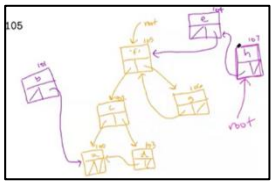

```
rope_iterator& rope_iterator::operator++()
{
```

WORKING MEMORY

| address | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
|---|---|---|---|---|---|---|---|---|
| value | a | b | c | d | e | f | g | h |
| left | 0 | 0 | 100 | 100 | 0 | 102 | 105 | 104 |
| right | 0 | 100 | 103 | 0 | 105 | 106 | 0 | 0 |

COPY MEMORY

| address | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 |
|---|---|---|---|---|---|---|---|---|
| value | f | c | g | a | d | | | |
| left | 102 | 100 | 105 | 0 | 111 | | | |
| right | 106 | 103 | 0 | 0 | 0 | | | |




| address | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
|---|---|---|---|---|---|---|---|---|
| value | a | b | c | d | e | f | g | h |
| left | 0 | 0 | 100 | 100 | 0 | 102 | 105 | 104 |
| right | 0 | 100 | 103 | 0 | 105 | 106 | 0 | 0 |
| marks | ✗ | o | ✗ | ✗ | o | ✗ | ✗ | o |

root: 105
free:
stack:

```
void pop_max() {
  assert(!m_heap.empty());
  int tmp = (size()+1)/2;
  for (int i = tmp+1; i < size(); i++) {
    if (m_heap[tmp] < m_heap[i])
      tmp = i;
  }
  m_heap[tmp] = m_heap.back();
  m_heap.pop_back();
  this->percolate_up(tmp);
}
```

```
void priority_queue_sort(priority_queue<T, vector<T>, greater<T>>& pq,
{
    output_count = 0;
    for (unsigned int i = pq.size() - 1; i >= 0; i--) {
        ostr<<pq.top()<<"\n"; //add the top value
        output_count++;
        if (pq.size() > 1)
            pq.pop(); //remove the top value to access the next value
        else
            break; //no values left to add
    }
}
```

**Friend Function**: It is basically a function that is used to access all private and protected members of classes. It is considered as a non-member function of class and is declared by the class that is granting access. This function is prefixed using the friend keyword in the declaration as shown

map find() Returns an iterator to the element with key-value 'g' in the map if found, else returns the iterator to end

begin() – Returns an iterator to the first element in the map.
end() – Returns an iterator to the theoretical element that follows the last element in the map.
size() – Returns the number of elements in the map.
max_size() – Returns the maximum number of elements that the map can hold.
empty() – Returns whether the map is empty.
pair insert(keyvalue, mapvalue) – Adds a new element to the map.
erase(iterator position) – Removes the element at the position pointed by the iterator.
erase(const g)– Removes the key-value 'g' from the map.
clear() – Removes all the elements from the map.

```
single element (1)   pair<iterator,bool> insert (const value_type& val);
with hint (2)        iterator insert (iterator position, const value_type& val);
```

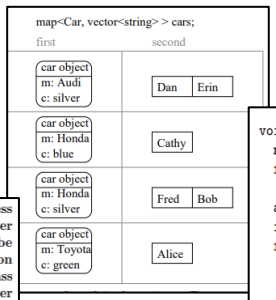
```
std::string token;
while (std::cin >> token)
```

```
ifstream file(font_file);
string str;
while (getline(file, str))
```

in .CPP file

```
ostream& operator<<(ostream &ostr, const Office &o) {
  ostr << "The " << o._name << " office has "
       << o._num_desks << " desks:" << endl;
  for (int i = 0; i < o._num_desks; i++) {
    ostr << "  desk[" << i << "] = " << o._desks[i] << endl;
  }
  return ostr;
}
```

Solution: This operator has been implemented as a friend function, which is necessary to gain access to the private member variables of the `Office` object. If we had implemented it as a non-member function the function wouldn't have access to these variables and accessor functions would need to be added to the public interface, which is undesirable since that would expose the private representation unnecessarily. We cannot implement the stream operator as a member function of the `Office` class because the << syntax requires the `ostream` object to be the first argument of the operator. In order to be written as a member operator of a particular class, the first argument must be of that class type.




```
Office::~Office() {
  for (int i = 0; i < _num_desks; i++) {
    if (_desks[i] != "") {
      _unassigned.push(_desks[i]);
    }
  }
  delete [] _desks;
}
```

don't need to delete each memory block, just delete pointer to array

```
void even_array(const list<int>& b, int* & a, int& n) {
  n = 0;    // count
  for (list<int>::const_iterator p = b.begin(); p!=b.end(); ++p)
    if (*p % 2 == 0) ++n;
  a = new int[n];   // allocate
  int* q = a;     // store in array
  for (list<int>::const_iterator p = b.begin(); p!=b.end(); ++p)
    if (*p % 2 == 0) {
      *q = *p;
      ++q;
    }
}
```

now subscripting dynamically accessed into

```
template <class T>
void cs2list<T>::splice(iterator itr, cs2list<T>& second) {
  if (second.empty()) return;
  second.head_->prev_ = itr.ptr_;
  second.tail_->next_ = itr.ptr_->next_;
  if (itr.ptr_->next_) {
    itr.ptr_->next_->prev_ = second.tail_;
  } else { // itr.ptr_ is the tail, so it must be reset
    this->tail_ = second.tail_;
  }
  itr.ptr_->next_ = second.head_;
  this->size_ += second.size_;
  second .size_ = 0;
  second.head_ = second.tail_ = 0;
}
```

empty second list

```
bool operator<(const Car &a, const Car &b) {
  return (a.getMaker() < b.getMaker() ||
          (a.getMaker() == b.getMaker() && a.getColor() < b.getColor()));
}
```

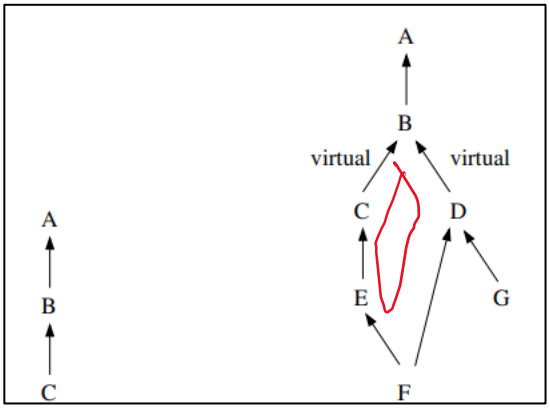
necessary to be written in non-member

```
class A {
public:
  virtual void f() { cout << "A::f\n"; }
  void g() { cout << "A::g\n"; }
};

class B : public A {
public:
  void g() { cout << "B::g\n"; }
};

class C : public B {
public:
  void f() { cout << "C::f\n"; }
  void g() { cout << "C::g\n"; }
};

int main() {
  A* a[3];
  a[0] = new A();
  a[1] = new B();
  a[2] = new C();

  for (int i = 0; i < 3; i++) {
    cout << i << endl;
    a[i]->f();
    B* b = dynamic_cast<B*>(a[i]);
    if (b) b->g();
  }
}
```

```
0
A::f
1
A::f
B::g
2
C::f
B::g
```

Inheritance

```
bool EqualsChildrenSum(Node *node) {
  if (node == NULL) return true;
  if (node->left == NULL && node->middle == NULL && node->right == NULL)
    return true;
  int sum = 0;
  if (node->left != NULL) sum += node->left->value;
  if (node->middle != NULL) sum += node->middle->value;
  if (node->right != NULL) sum += node->right->value;
  if (sum != node->value) return false;
  return
    EqualsChildrenSum(node->left) &&
    EqualsChildrenSum(node->middle) &&
    EqualsChildrenSum(node->right);
}
```




- As a result, for each subscript, $i$,
  - The parent, if it exists, is at location $\lfloor (i-1)/2 \rfloor$.
  - The left child, if it exists, is at location $2i+1$.
  - The right child, if it exists, is at location $2i+2$.

```
Storage::~Storage() {
  for (int w = 0; w < width; w++) {
    for (int d = 0; d < depth; d++) {
      for (int h = 0; h < height; h++) {
        if (data[w][d][h] != NULL) {
          remove(data[w][d][h],w,d,h);
        }
      }
      delete [] data[w][d];
    }
    delete [] data[w];
  }
  delete [] data;
}
```

A) vector B) list C) map D) set
E) priority queue F) hash table

**Solution:**

| | |
|---|---|
| **B C D** | allows efficient (sublinear) removal of the first and last elements (or the minimum and maximum elements) |
| **A E F** | uses an array or vector as the underlying representation |
| **B C D (F)** | uses a network of nodes connected by pointers as the underlying representation |
| **C D (E) F** | the underlying data structure must be "balanced" or well-distributed to achieve the targeted performance |
| **C D E** | requires definition of `operator<` or `operator>` |
| **C D E F** | entries cannot be modified after they are inserted (requires re-insertion or re-processing of position) |
| **C D (F)** | duplicates are not allowed |
| **B** | allows sublinear merging of two of instances of this data structure |

| | |
|---|---|
| **Solution: VECTOR** | Knows how many elements it contains. |
| **Solution: BOTH** | Can be used to store elements of any type. |
| **Solution: NEITHER** | Prevents access of memory beyond its bounds. |
| **Solution: VECTOR** | Is dynamically re-sizable. |
| **Solution: BOTH** | Can be passed by reference. |

Prio-Queue

| Function | Time Complexity | Space Complexity |
|---|---|---|
| Q.top() | O(1) | O(1) |
| Q.push() | O(log n) | O(1) |
| Q.pop() | O(log n) | O(1) |
| Q.empty() | O(1) | O(1) |

Map/Set

| Function | Time Complexity | Space Complexity |
|---|---|---|
| M.find(x) | O(log n) | O(1) |
| M.insert(pair<int, int> (x, y) | O(log n) | O(1) |
| M.erase(x) | O(log n) | O(1) |
| M.empty( ) | O(1) | O(1) |
| M.clear( ) | Theta(n) | O(1) |
| M.size( ) | O(1) | O(1) |

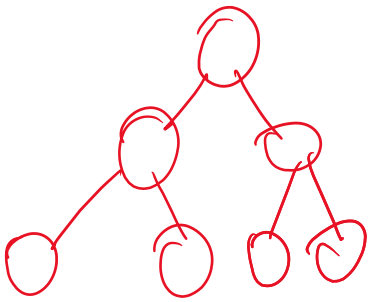
Prio-Queue defaults to max_heap need to add greater<> for min heap

| | |
|---|---|
| priority_queue::empty() | Returns whether the queue is empty. |
| priority_queue::size() | Returns the size of the queue. |
| priority_queue::top() | Returns a reference to the topmost element of the queue. |
| priority_queue::push() | Adds the element 'g' at the end of the queue. |
| priority_queue::pop() | Deletes the first element of the queue. |
| priority_queue::swap() | Used to swap the contents of two queues provided the queues must be of the same type, although sizes may differ. |
| priority_queue::emplace() | Used to insert a new element into the priority queue container. |
| priority_queue value_type | Represents the type of object stored as an element in a priority_queue. It acts as a synonym for the template parameter. |

- As a result, for each subscript, $i$,
  - The parent, if it exists, is at location $\lfloor (i-1)/2 \rfloor$.
  - The left child, if it exists, is at location $2i+1$.
  - The right child, if it exists, is at location $2i+2$.

```
private:
  // HELPER FUNCTIONS
  void percolate_up(int i) {
    T value = m_heap[i];
    while (i > 0) {
      int parent = (i-1)/2;
      if (value >= m_heap[parent]) break;  // done
      m_heap[i] = m_heap[parent];
      i = parent;
    }
    m_heap[i] = value;
  }
  void percolate_down(int i) {
    T value = m_heap[i];
    int last_non_leaf = int(m_heap.size()-1)/2;
    while (i <= last_non_leaf) {
      int child = 2*i+1, rchild = 2*i+2;
      if (rchild < m_heap.size() && m_heap[child] > m_heap[rchild])
        child = rchild;
      if (m_heap[child] >= value) break;  // found right location
      m_heap[i] = m_heap[child];
      i = child;
    }
    m_heap[i] = value;
  }

  // REPRESENTATION
  vector<T> m_heap;
```

Prio Queue can have duplicates



→ all leaf nodes must be filled until move onto next row going from left-right

Stack

empty()
size() –
top() – R
push(g)
pop() –

↳ same for queue except front() back() no top()

```
hash ^= ((hash << 5) + totalQuery[i] + (hash >> 2));
```