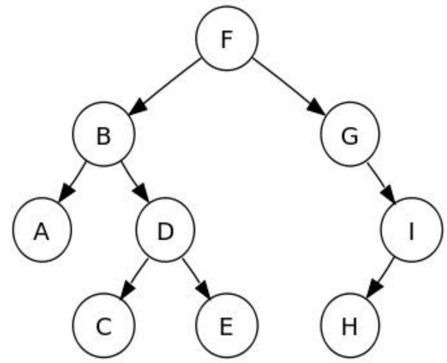
std::sort(itr1, itr2[, sort_function]) (for vector)
Array:
Type array[size]: int a[5]
Array[index] = element: a[1] = 2
Vector:
O(1) std::vector<type> name;
O(1) name.push_back(element);
O(1) vector.pop_back() remove the last value;
O(n) std::vector<type> name(size, fill_value);
O(n) std::vector<type> copy(name_itr_1, name_itr_2);
O(n) std::vector<type> copy(name);
O(1) vector.size() return size
O(1) vector.back() return the last elemnt;
O(1) vector.front() return the first element;
O(n) vector.insert(itr, value) return itr of the first inserted value
O(n) vector.erase(itr) remove element at iterator itr
return the iterator next to the itr
vector.empty()
List:
There is NO pop_front() in vectors!
Iterators in lists could not do itr + 5 erase(iterator)
Returns an itr pointing after the erased element
insert(iterator, new element)
Add the new element before the itr
Returns an itr pointing at the new element
O(1) list<type> name;
O(n) list<type> name(itr_1, itr_2)
O(1) list.erase(itr) remove itr and return the next itr
O(1) list.insert(itr, value)
O(n) list.remove(value) remove all value from list
O(1) list.pop_back(value)/pop_front(value)
O(1) list.push_back(value)/push_front(value)
O(nlogn)list.sort()
Set
O(1) set<type, compare> s;
O(nlogn)set<type, compare> s(itr_1,itr_2);
O(log n) set.insert(key) insert key to set and return pair<itr, bool>
itr is pos of key, and bool = true if inserted = false if exist
O(log n) set.insert(itr, key) itr is a hint, but key will go correct pos
Return itr where key goes
O(log n)set.find(key) return itr of key, .end() if not exist
O(log n)set.erase(key) return number of key erased(1 or 0)
O(log n)set.erase(itr) no return
O(log n)set.erase(itr_1, itr_2) no return
Map
itr -> first key itr -> second value
O(1) map<ket_type, value_type> m;
O(nlog n)map<key_type, value_type> m(itr_1, itr_2);
O(log n) m.find(key) return map itr; .end() if not exist
O(log n)m.insert(std::make_pair(key, value)); return pair<itr, bool>
O(log n) erase(itr) erase the pair referred by itr no return
O(log n) erase(key) erase pair containing the key k, return 0 or 1
Pair
Pair.first pair.second
std::pair<type1, type2> p1(n,m)e.g. std::pair<int, double> p1(5,7.5)
std::pair<type1, type2> p2 = std::make_pair(8,9.5)

Section 13, 4:00 – 5:50 pm
TA: Stephane
Mentors: Wilson, Casey, Terry, Tyler

Binary tree:



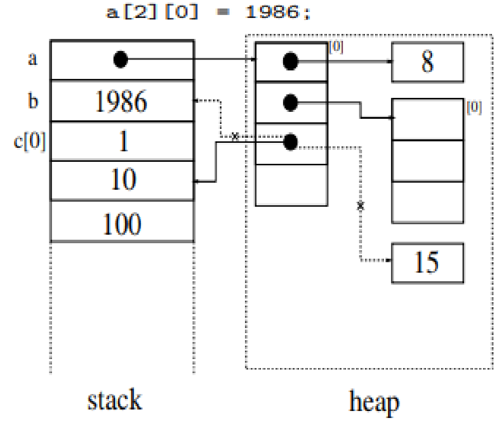
Depth-first

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

```
class Node{ public:
  Node* left;
  Node* right;
  char val; };
void printPostOrder(Node* node){
  if (node == NULL){ return;}
  printPostOrder(node->left);
  printPostOrder(node->right);
  cout << node->val;}
void printInOrder(Node* node){
  if (node == NULL) {return;}
  printInOrder(node->left);
  cout << node->val;
  printInOrder(node->right);}
void printPreOrder(Node* node){
  if (node == NULL) {return;}
  cout << node->val;
  printPreOrder(node->left)
  printPreOrder(node->right);}
```

```
int **a;
a = new int*[4];
a[0] = new int;
a[1] = new int[3];
a[2] = new int;
int b = 25;
int c[3] = {1,10,100};

*(a[2]) = 15;
a[0][0] = 8;
a[2] = &b;
a[2][0] = 1986;
```



Binary Search

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
  if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low, mid, x);
  else
    return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
  return binsearch(v, 0, v.size()-1, x);
}
```

```
class Polygon {
public:
  Polygon() {}
  virtual ~Polygon() {}
  int NumVerts() { return verts.size(); }
  virtual double Area() = 0;
  virtual bool IsSquare() { return false; }
protected:
  vector<Point> verts;
};

class Triangle : public Polygon {
public:
  Triangle(Point pts[3]) {
    for (int i = 0; i < 3; i++) verts.push_back(pts[i]); }
  double Area();
};

class Quadrilateral : public Polygon {
public:
  Quadrilateral(Point pts[4]) {
    for (int i = 0; i < 4; i++) verts.push_back(pts[i]); }
  double Area();
  double LongerDiagonal();
  bool IsSquare() { return (SidesEqual() && AnglesEqual()); }
private:
  bool SidesEqual();
  bool AnglesEqual();
```

std::for_each(begin_itr, end_itr, functor_called_itr)
e.g. std::for_each(my_data.begin(), my_data.end(), float_print);

std::unordered_map<std::string,Foo> m; using default hash function
std::unordered_map<std::string,Foo,MyHashFunctor> m(1000);
using own hash function

Stacks allow access, insertion and deletion from only one end called the *top*

* There is no access to values in the middle of a stack.
* Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
* All stack operations are $O(1)$

Queues allow insertion at one end, called the *back* and removal from the other end, called the *front*

* There is no access to values in the middle of a queue.
* Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
* All queue operations are $O(1)$

priority queue(binary heap)
main operator: insert/push/pop
have a specific order.
NODE: O(log n)

- Functions that are common, at least have a common interface, are in Polygon.
- Some of these functions are marked virtual, which means that when they are redefined by a derived class, this new definition will be used, even for pointers to base class objects.
- Some of these virtual functions, those whose declarations are followed by = 0 are *pure virtual*, which means they must be redefined in a derived class.
  - Any class that has pure virtual functions is called "abstract".
  - Objects of abstract types may not be created — only pointers to these objects may be created.
- Functions that are specific to a particular object type are declared in the derived class prototype.

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    //  Choose the child to compare against
    if (p->right && p->right->value < p->left->value)
      child = p->right;
    else
      child = p->left;
    if (child->value < p->value) {
      swap(child, p); // value and other non-pointer member vars
      p = child;
    }
    else
      break;
  }
```

```
percolate_up(TreeNode<T> * p) {
  while (p->parent)
    if (p->value < p->parent->value) {
      swap(p, parent);  // value and other non-pointer member vars
      p = p->parent;
    }
    else
      break;
}
```

```
void percolate_up(){
      for(unsigned int child = m_heap.size()-1; child > 0;)
              int parent = (child-1)/2;
      if (m_heap[child] < m_heap[parent]){
          T tmp = m_heap[child];
          m_heap[child] = m_heap[parent];
          m_heap[parent] = tmp;
          child = parent;
      }
      else{
              child--;
      }
  }
}
```

- 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 $2i+2$.

```
void percolate_down(){
      int parent = 0;
  while ((parent*2)+1 < m_heap.size()){
      int left_child = (parent*2)+1;
      int right_child = left_child+1;
      int swap = parent;
      if(m_heap[swap] > m_heap[left_child]){
              swap = left_child;
      }
      if(right_child < m_heap.size() && m_heap[swap] > m_heap[right_child]){
              swap = right_child;
      }
      if (swap != parent){
          T tmp = m_heap[parent];
          m_heap[parent] = m_heap[swap];
          m_heap[swap] = tmp;
          parent = swap;
      }
      else{
              parent++;
      }
  }
}
```

inheritance:

```
class Dragon: virtual public Pokemon{
  public:
    Dragon(const std::map<std::string,std::vector<std::string> > &facts);
class Charmander: public Dragon, public Monster{
  public:
    Charmander(const std::map<std::string,std::vector<std::string> > &facts);
};
class Charmeleon: public Charmander{
  public:
    Charmeleon(const std::map<std::string,std::vector<std::string> > &facts);
};
```

```
Dragon::Dragon(const std::map<std::string,std::vector<std::string> > &facts): Pokemon(facts){
  if( (this->getEggGroups()).size() != 2){
    throw 1;
Charmander::Charmander(const std::map<std::string,std::vector<std::string> > &facts): Pokemon(facts), Dragon(facts), Monster(facts){
Charmeleon::Charmeleon(const std::map<std::string,std::vector<std::string> > &facts): Pokemon(facts), Charmander(facts){
```

Exception:

```
int my_func(int a, int b) throw(double,bool) {
  if (a > b)
    throw 20.3;
  else
    throw false;
}

int main() {
  try {
    my_func(1,2);
  }
  catch (double x) {
    std::cout << " caught a double " << x << std::endl;
  }
  catch (...) {
    std::cout << " caught some other type " << std::endl;
  }
}
```

Garbage collection:

- **Reference Counting:**
  - + fast and incremental
  - − can't handle cyclical data structures!
  - ? requires ~33% extra memory (1 integer per node)
- **Stop & Copy:**
  - − requires a long pause in program execution
  - + can handle cyclical data structures!
  - − requires 100% extra memory (you can only use half the memory)
  - + runs fast if most of the memory is garbage (it only touches the nodes reachable from the root)
  - + data is clustered together and memory is "de-fragmented"

没有碎片

- **Mark-Sweep:**
  - − requires a long pause in program execution
  - + can handle cyclical data structures!
  - + requires ~1% extra memory (just one bit per node)
  - − runs the same speed regardless of how much of memory is garbage. It must touch all nodes in the mark phase, and must link together all garbage nodes into a free list.

清除阶段残留大量碎片

smart pointer

- `auto_ptr`
  When "copied" (copy constructor), the new object takes ownership and the old object is now empty. *Deprecated in new C++ standard.*
- `unique_ptr`
  Cannot be copied (copy constructor not public). Can only be "moved" to transfer ownership. Explicit ownership transfer. *Intended to replace* `auto_ptr`. std::unique_ptr has memory overhead only if you provide it with some non-trivial deleter. It has time overhead only during constructor (if it has to copy the provided deleter) and during destructor (to destroy the owned object).
- `scoped_ptr` (Boost)
  "Remembers" to delete things when they go out of scope. Alternate to `auto_ptr`. Cannot be copied.
- `shared_ptr`
  Reference counted ownership of pointer. Unfortunately, circular references are still a problem. Different subflavors based on where the counter is stored in memory relative to the object, e.g., `intrusive_ptr`, which is more memory efficient. std::unique_ptr has memory overhead only if you provide it with some non-trivial deleter. It has time overhead in constructor (to create the reference counter), in destructor (to decrement the reference counter and possibly destroy the object) and in assignment operator (to increment the reference counter).
- `weak_ptr`
  Use with `shared_ptr`. Memory is destroyed when no more `shared_ptr`s are pointing to object. So each time a `weak_ptr` is used you should first "lock" the data by creating a `shared_ptr`.
- `scoped_array` and `shared_array` (Boost)

```
bool search_from_loc(loc position, //  current position
                     const vector<string>& board,
                     const string& word,
                     vector<loc>& path)  //  path up to the current pos
{
  // DOUBLE CHECKING OUR LOGIC: the letter at the current board
  // position should equal the next letter in the word
  assert (board[position.row][position.col] == word[path.size()]);

  // start by adding this location to the path
  path.push_back(position);

  // BASE CASE: if the path length matches the word length, we're done!
  if (path.size() == word.size()) return true;

  //  search all the places you can get to in one step
  for (int i = position.row-1; i <= position.row+1; i++) {
    for (int j = position.col-1; j <= position.col+1; j++) {

      // don't walk off the board though!
      if (i < 0 || i >= int(board.size())) continue;
      if (j < 0 || j >= int(board[0].size())) continue;
      // don't consider locations already on our path
      if (on_path(loc(i,j),path)) continue;

      // if this letter matches, recurse!
      if (word[path.size()] == board[i][j]) {
        // if we find the remaining substring, we're done!
        if (search_from_loc (loc(i,j),board,word,path))
          return true;
      }
    }
  }
```

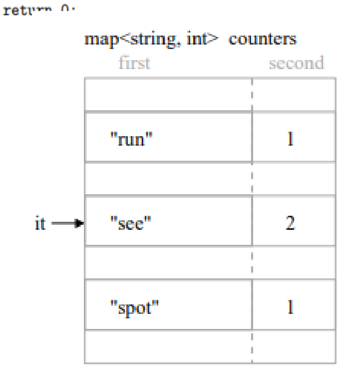
- Here's a simple and elegant solution to this problem using a map:

```
#include <iostream>
#include <map>
#include <string>

int main() {
  std::string s;
  std::map<std::string, int> counters; // store each word and an associated counter

  // read the input, keeping track of each word and how often we see it
  while (std::cin >> s)
    ++counters[s];

  // write the words and associated counts
  std::map<std::string, int>::const_iterator it;
  for (it = counters.begin(); it != counters.end(); ++it) {
    std::cout << it->first << "\t" << it->second << std::endl;
  }
  return 0;
```



Leftist Heap – implemented with trees

- Able to merge two heaps in O(log n) time
- Heap-order property
  - parent's priority value is <= to children's
  - result: minimum element is at the root