declarations (l-values)

```
int x; int a,b,c; int x=235;
short s; long l;
char c = 'a';
unsigned char u= 255; signed char s=-1;
float f; double d;
int a[] = {0,1,2}; int a[2][3] = {{1,2,3},{4,5,6}};
char s[] = "hello";
```

files (I/O) #include <fstream>

```
std::ifstream in_str("filename");
if (!in_str.good()) {
    std::cerr << "can't open \n";
    exit(1);
}
in_str >>x;
out_str << x <<y << std::endl;
```

std::setpercision(#) # of decimals
std::fixed forces decimal
std::setw(#) set aside width # output

std::ifstream varName(file) >> reading
std::ofstream varName(file) << writing

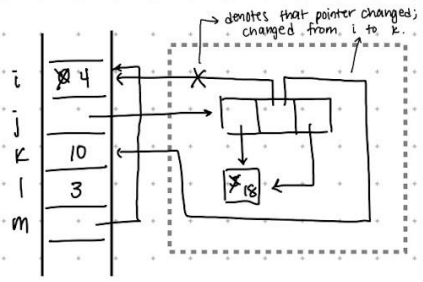
Strings

```
#include <string>
std::String s1, s2= "hello"
s1.size(), s2.size()
s1 += s2 + ' ' + "world";
s1.substr(m,n)
```

```
#include <cctype>
String.isalpha();   String.isdigit();
String.islower();   String.isupper();
String.tolower();   String.toupper();
```

POINTERS & MEMORY DIAGRAMS

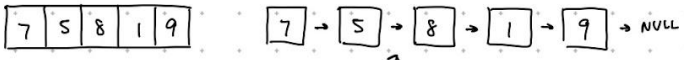
```
int i,**j,k,l,*m;          // "pointer to another pointer"; also normal; normal; just a pointer
i = 0;                     // INITIALIZED IN STACK
j = new int*[3];           // DECLARES AN ARRAY IN THE HEAP
j[0] = new int;            // DECLARES NEW POINTER INT IN HEAP FROM j[0]
j[1] = &i;                 // REFERENCE TO i
m = *(j+1);                // SAME AS SAYING j[1] REFERENCE m to j[1], which is to i ∴ m to i
j[1] = &k;
k=10;
*(j[0]) = 5;               // j[0] CHANGED
j[2] = j[0];
*(j[0]) = 18;
*m = 4;
l = 3;                     // JUST AN 1.
```




```
delete[] j;   //ARRAYS
delete [insert variable];
```

ARRAYS: stores a fixed-size of elements; size cannot be changed or copied
ONLY A POINTER → cannot determine size (no use of sizeof())
cannot be returned from a function (unless std::array)

INITIALIZATION: `int array_name[n];` ← n is predetermined array size



ITERATORS: identifies a container & element stored there
provides operation for moving between elements
restricts operations to what the container can handle

BIG O NOTATION

	VECTORS	SINGLE LISTS	DOUBLE LISTS	SETS/MAPS
size()	O(1)	O(n)	O(1)	O(1)
push_back()	O(1)	O(n)	O(1)	O(1)
erase()	O(n)	O(1)	O(1)	O(logn)
insert()	O(n)	O(1)	O(1)	O(logn)
pop_back()	O(1)	O(n)	O(1)	O(1)
find() SORTED:	O(logn)	O(n)	O(logn)	O(logn)
find() OTHER:	O(n)	O(n)	O(n)	O(logn)

```
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);
}
```

RECURSION

rules: 1. Always have a base case
2. Define solution in terms of smaller problems
3 Remember corner cases
4. Figure it out & Good luck!

* All RECURSIVE CALLS SHOULD PROGRESS TO BASE CASES

vector #include <vector>

```
vector<int> a(10);
a.size();
a.insert(itr, x)
a.push_back;
a.pop_back();
a.erase(itr pos)
a.front(); a.back()=4;
a[20]=1;
for(int& p : a) { p=0;}
vector<int>::iterator p = a.begin();
for(; p != a.end(); ++p) { *p=0; }
```

list #include <list>

```
list<int> b(10);
b.size();
b.insert(itr, x);
b.push_back(); b.push_front();
b.pop_back(); b.pop_front();
b.erase(itr pos);
b.front(); b.back();

list<int>::iterator p = b.begin();
for(; p != b.end(); ++p)
  { *p = 0; }
```

map #include <map>

```
map<string,int> a;
a["hello"] = 3;
a.first; a.second;
a.size()

a.insert(key) = definition
itr = a.find(key)
```

set #include <set>

```
set<int> s;
s.insert(123);
if (s.find(123) != s.end())
    s.erase(123);
```

iterators

```
a.begin(); a.end();      NORMAL
a.rbegin(); a.rend();    REVERSE
a.cbegin(); a.cend();    CONST NORMAL
a.crbegin(); a.crend();  CONST REVERSE
```

others

```
min(x,y); max(x,y);
swap(x,y);
sort(a, a+n);
sort(a.begin(), a.end());
reverse(a.begin(), a.end());
```

// VECTOR REVERSE

```
template <class T>
void reverse(std::vector<T> &data) {
  int end = data.size()-1;
  if (end == 0) {return; }
  else if (end == 1) {
    T initial = data[0];
    data[0] = data[1];
    data[1] = initial;
    return;
  }
  for (unsigned int i = 0; i < data.size()/2; i++) {
    T initial = data[i];
    data[i] = data[end-i];
    data[end-i] = initial;
  }
}
```

//POINTER REVERSE

```
template <class T>
void reverse(Node<T>* &input) {
  Node<T>* prev = NULL;
  Node<T>* next = NULL;
  Node<T>* current = input;
  while (current != NULL) {
    next = current -> ptr;  //2
    current -> ptr = prev;
    prev = current;
    current = next;
  }
  input = prev;
}
```

//ITERATOR REVERSE

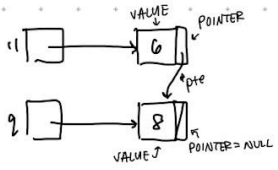
```
template <class T>
void reverse(std::list<T> &data) {
  typename std::list<T>::iterator itr = data.begin();
  typename std::list<T>::reverse_iterator ritr = data.rbegin();
  int size = data.size();
  int counter = 0;

  while (counter != size/2) {
    T initial = *itr;
    *itr = *ritr;
    *ritr = initial;
    counter++;
    ++ritr; ++itr;
  }
}
```

//using erase & push_front

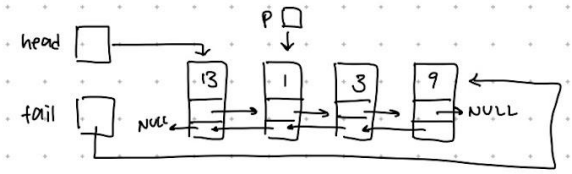
```
template <class T>
void reverse(std::list<T> &data) {
    typename std::list<T>::iterator itr = data.begin();
    for (int i = 0; i < data.size(); ++i) {
        T initial = *itr;
        itr = data.erase(itr);
        data.push_front(initial);
    }
}
```

NODE & CLASSES

```
template <class T>
class Node {
public:
   T value;
   Node* pte;
};
```

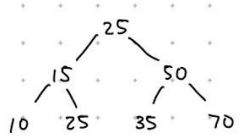



```
int main() {
   Node<int>* 11;          //ll is a pointer to a Node
   ll = new Node<int>      // create a Node assign address ot ll
   ll->value = 6           // (*ll) = 6;
   ll->ptr = NULL          // tail to a NULL

   Node<int>* q = new Node<int>;
   q->value = 8;
   q->ptr = NULL;

   ll->ptr = q             // set ll's pointer to point to q
}
```



DOUBLE LINKED LISTS ARE VERY SIMILAR!!

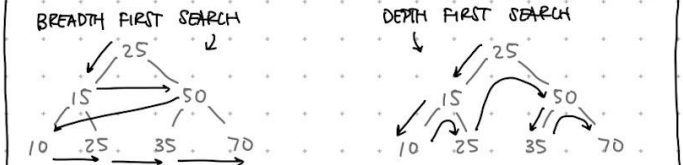
```
template <class T>
class Node {
public:
   T value;
   Node* next;
   Node* prev;
};
```



INORDER: (LEFT ROOT RIGHT) DFS
10, 15, 22, 25, 50, 35, 70

PRE-ORDER: (ROOT LEFT RIGHT) DFS
25, 15, 10, 22, 50, 35, 70

POST-ORDER: (LEFT RIGH ROOT) DFS
10, 22, 15, 85, 70, 50, 25



PROS: finds solution node w/ shortest path to root

CONS: memory intensive b/c it needs to store all current nodes

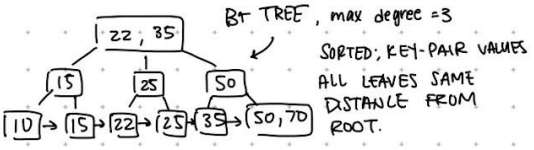
PROS: quickly searches leaf nodes

CONS: if it makes an "incorrect" branch decision, it will take a long time.

(for regular tree)

```
// PRIVATE HELPER FUNCTIONS
TreeNode<T>*  copy_tree(TreeNode<T>* old_root) {
  if (old_root == NULL)
    return NULL;
  TreeNode<T> *answer = new TreeNode<T>();
  answer->value = old_root->value;
  answer->left = copy_tree(old_root->left);
  answer->right = copy_tree(old_root->right);
  return answer;
}

void destroy_tree(TreeNode<T>* p) {
  if (!p) return;
  destroy_tree(p->right);
  destroy_tree(p->left);
  delete p;
}

iterator find(const T& key_value, TreeNode<T>* p) {
  if (!p) return iterator(NULL);
  if (p->value > key_value)
    return find(key_value, p->left);
  else if (p->value < key_value)
    return find(key_value, p->right);
  else
    return iterator(p);
}

int erase(T const& key_value, TreeNode<T>* &p) {
  if (!p) return 0;

  // look left & right
 if (p->value < key_value)
    return erase(key_value, p->right);
  else if (p->value > key_value)
    return erase(key_value, p->left);

  // Found the node.  Let's delete it
  assert (p->value == key_value);
  if (!p->left && !p->right) { // leaf
    delete p; p=NULL;
  } else if (!p->left) { // no left child
    TreeNode<T>* q = p; p=p->right; delete q;
  } else if (!p->right) { // no right child
    TreeNode<T>* q = p; p=p->left; delete q;
  } else { // Find rightmost node in left subtree
    TreeNode<T>* &q = p->left;
    while (q->right) q = q->right;
    p->value = q->value;
    int check = erase(q->value, q);
    assert (check == 1);
  }
  return 1;
}
```

(for (B+ tree)



B+ TREE, max degree = 3

SORTED, KEY-PAIR VALUES

ALL LEAVES SAME DISTANCE FROM ROOT.

```
void print_sideways(std::ostream& out_str, BPlusTreeNode<T>* p, int depth) {
  if (!p) {
      out_str << "Tree is empty." << std::endl;
      return;
  }
  else if (!p->is_leaf()) {
      for (uint c = 0; c < p->children.size(); ++c) {
          print_sideways(out_str, p->children[c], depth + 1);
          if (c == 0) { print_format(out_str, depth, p); }
      }
  }
  else if (p->is_leaf()) { print_format(out_str, depth, p); }
}

void print_BFS(std::ostream& out_str, BPlusTreeNode<T>* p) {
    std::vector<BPlusTreeNode<T>*> current, next;
    if (!p) {
        out_str << "Tree is empty." << std::endl;
        return;
    }

    current.push_back(p);
    while (current.size() != 0) {
        for (uint i = 0; i < current.size(); ++i) {
            BPlusTreeNode<T>* branch = current[i];
            for (uint c = 0; c < branch->children.size(); ++c) {
                next.push_back(branch->children[c]);
            }

            for (uint k = 0; k < branch->keys.size(); ++k) {
                if (k == 0) {
                    out_str << branch->keys[k];
                    continue;
                }
                out_str << "," << branch->keys[k];
            }

            if ( i != current.size() - 1) { out_str << "\t";}
        }
        out_str << "\n";
        current = next;
        next.clear();
    }
}
```

```
//driver function
bool insert(int val, TreeNode*&p){
    TreeNode* start = p;
    //get the leftmost element
    while(start && start->left)
        start = start->left;
    return insert(val,p,start);
}

bool insert(int val, TreeNode*& p, TreeNode* first, TreeNode* prev = NULL){
    //we've reached a leaf node
    if(!p){
        //set p to a new node (passed by reference, so parent auto updates)
        p = new TreeNode(val);
        //this means this element is not the first, so we can just use prev's next
        if(prev){
            p->next = prev->next;
            prev->next = p;
        }
        else //otherwise, p's next is the first element
            p->next = first;
        return true;
    }
    else if (val < p->value) //if we go left, prev should not be changed
      return insert(val, p->left, first, prev);
    else if (val > p->value) //if we go right, prev should be p
      return insert(val, p->right, first, p);
    else //element already exists in the set
      return false;
}
```

Operator Overload:

- Athematic operations: + – * / %
  - Class Class::operator+(const Class &c)
  - Class operator+(const Class &c, const Class &c2)
- Relational operations: == != >= <=
  - bool Class::operator<(const Class &c)
  - bool operator<(const Class &c1, const Class &c2)
- Stream operations << >> (replace with istream when needed)
  - ostream& Class::operator<<(ostream& outStr)
  - ostream& operator<<(ostream& outStr, Class &c)

Solution:

```
void findBoxes(const DonutBox& box, DonutBox& current_box, std::vector<DonutBox>& boxes){
    if(box.empty()){
        boxes.push_back(current_box);
        return;
    }
    for(unsigned int i=0; i<box.size(); i++){
        DonutBox tmp_box = box;
        current_box.push_back(box[i]);
        tmp_box.erase(tmp_box.begin()+i);
        findBoxes(tmp_box, current_box, boxes);
        current_box.pop_back();
    }
}

void findBoxes(const DonutBox& box, std::vector<DonutBox>& boxes){
    DonutBox tmp;
    findBoxes(box, tmp, boxes);
}
```