Doubly linked lists

- internal nodes have pointers to “previous” and “next” neighbours in the list
- overall list maintains pointers to the first and last (or front and back) nodes in the list
- first will consider a struct based approach (will do classes in ~2 weeks)
- use one struct to define the individual nodes
- use another struct for the overall list
- functions to insert, remove, search, print, delete, etc

Node struct

- various data fields for the information you want stored for each node
- pointers to previous and next nodes in list (null when there is no previous/next, i.e. for the nodes at either end)

```
struct node {
    float somedata;
    int moredata;
    node* prev;
    node* next;
};
```

List struct

- maintain info about the list as a whole
- might be simply pointers to the front/back nodes

```
struct List {
   node* front;
   node* back;
};
```

access functions

- most functions will simply be passed a List variable (by ref if needed) and any needed data (e.g. value to search for)
  - insert info into a list (at front? at back? at specific spot?)
  - search a list for a specific value
  - print a list
  - remove from the list (from front? back? inside?)
  - deallocate all the list content

initializing a list

- before we insert anything, to show list is empty, set front/back to null

```
void initialize(List &L)
{
   L.front = NULL;
   L.back = NULL;
}

int main()
{
   List list1, anotherlist;
   initialize(list1);
   initialize(anotherlist);
   ...
```

creating a new node

- given the data values to use, create and initialize the node
- initialize next/prev to null by default?
- return pointer to the newly created node

```
node* create(int sd, float md)
{
   node* n = new node; // ** skipping error checking here for now
   n->somedata = sd;  // using -> since we're accessing struct field through a pointer
   n->moredata = md;
   n->prev = NULL;
   n->next = NULL;
   return n;
}
```

inserting a node at front or back

- create the node using the data values given
- connect with the list's old front/back
- update the list's front/back

```
void insertFront(List &L, int sd, float md)
{
   node* n = create(sd, md);
   if (n != NULL) {
      if (L.front == NULL) { // list used to be empty
         L.front = n;
         L.back = n;
      } else { // general case
         n->next = L.front;  // pointer from new node to old front
         L.front->prev = n;  // pointer from old front back to n
         L.front = n;        // update L to recognize new front of list
   }
}
```

removing from front/back

- reverse of insert: update pointers, front/back, delete

```
void removeFront(List &L)
{
   if (L.front != NULL) { // need to check since L might have been empty
      node* n = L.front;  // get ptr to node we're removing
      L.front = n->next;   // update front
      L.front->prev = NULL; // so new front knows nothing is before it
      delete n; // release the memory from the removed node
   }
}
```

searching for data values

- often have one function that finds a node with targetted value, returns pointer to the node (other functions can call this as needed)
- example: search for first node with specific value in somedata field

```
node* search(List L, int sd) // not changing L, doesn't need to be pass-by-ref
{
    node* n = L.front; // search node-by-node from front to back
    while (n != NULL) {
        if (n->somedata == sd) {
           return n;  // found it, leave now and return the ptr
        }
        n = n->next; // move on to next node in list
    }
    return NULL; // never found it, return NULL by default
}
```

lookup (using node* search)

- find node with a target somedata value (if any such node exists) and look up its associated moredata value
- return true if found, false otherwise

```
bool lookup(List L, int sd, float &md) // md pass by ref so we can fill it in
{
    node* n = search(L, sd);
    if (n == NULL) {
       return false; // didn't find it
    } else {
       md = n->moredata;
       return true;
    }
}
```

find and remove a specific node

- find the node then chop it from the list
- could be at front, back, or somewhere inside

```
bool remove(List &L, int sd)
{
    node* n = search(L, sd);
    if (n == NULL) {
        return false; // no such node found
    }
    if (n == L.front) { // we're removing front node
        removeFront(L); // just use the removeFront we created earlier
    } else if (n == L.back) { // we're removing back node
        removeBack(); // assuming we created a removeBack like our removeFront
    }
```

find & remove continued...

- the general case, now we know it's an internal node

```
  else {
      // find the nodes before and after the node being removed
      node* before = n->prev;
      node* after = n->next;
      // have them bypass n
      before->next = after;
      after->prev = before;
      // deallocate the old node
      delete n;
  }
  return true; // successfully removed
}
```

insert into sorted list

- suppose we want to maintain sorted list, e.g. suppose we keep values sorted by their somedata field (increasing from front of list to back)
- instead of insert at front/back, find the right spot to insert new nodes
- special cases for inserts at the ends
- general case, find the node that belongs before the new node, get the node after that one, insert “between” them

insertSorted

```
bool insertSorted(List &L, int sd, float md)
{
    // create the node to be inserted, return false if fails
    node* n = create(sd, md);
    if (n == NULL) {
       return false;
    }

    // special case 1: list used to be empty
    if (L.front == NULL) {
       L.front = n;
       L.back = n;
    }
```

insertSorted, general case

```
  else { // general case, belongs between two existing nodes somewhere in list
      // find the node that belongs after our new node,
      // i.e. the first node whose somedata field is >= sd
      node* after = L.front;
      while (sd > after->somedata) {
          after = after->next;
      }
      // now find the node before that one
      node* before = after->prev;
      // and insert n between the two of them
      n->prev = before;
      n->next = after;
      before->next = n;
      after->prev = n;
  }
  return true; // done!
}
```

insertSorted continue

```
// special case 2: belongs at front, i.e. sd < old front element's somedata field
else if (sd < L.front->somedata) {
   n->next = L.front;
   L.front->prev = n;
   L.front = n;
}

// special case 3: belongs at back (sd > old back's somedata)
else if (sd > L.back->somedata) {
    n->prev = L.back;
    L.back->next = n;
    L.back = n;
}
```