Linked lists
Data structures to store a collection of items

- Data structures to store a collection of items are commonly used
  - Typical operations on such data structures: insert, remove, find_max, update, etc

- What are our choices so far to design such a data structure?
Data structures to store a collection of items

- What are the choices so far?
  - Arrays
    - Limitations
      - fixed capacity, memory may not be fully utilized.
      - Insert and remove can be expensive (a lot of copies) if we don’t want to leave holes in the middle of the array.
      - Continuous memory for easy index
  - Dynamic arrays
    - Limitations:
      - Capacity is dynamic, memory still may not be fully utilized, but better than static arrays.
      - Insert and remove can be expensive, especially when the capacity changes.
      - Continuous memory for easy index
Linked list is another choice.
- A true dynamic data structure in that each item in the list is dynamically allocated using a new operator.
- Capacity is always the same as memory used (with tax)
- Insert and remove operations are cheap
- Memory are not continuous
- Limitations: no (or expensive) [] operator.

Linked list is one of the “linked data structures”.

Linked list and array

- An array of string:
  - \(S[0] = \text{“abc”};\)
  - \(S[1] = \text{“white”};\)
  - \(S[2] = \text{“black”};\)

- A linked list of strings
  - Each item has two fields
    - A string field
    - A pointer pointing to the next item.

```cpp
class listofstring {
public:
    string item;
    listofstring *next;
};
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“abc”</td>
</tr>
<tr>
<td>1</td>
<td>“white”</td>
</tr>
<tr>
<td>2</td>
<td>“black”</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
```

```
“abc”  “white”  “black”
```

```
head
```

```
“abc”  “white”  “black”
```

```
head
```

```
“abc”  “white”  “black”
```

```
head
```

```
“abc”  “white”  “black”
```

```
head
```
Linked list

- No waste of memory (except for pointers).
- Each box is dynamically allocated by a new operation.
- Several variations
  - Singly linked list
  - Doubly linked lists
A doubly linked list

- Let us assume that we store two data fields in each node: a string and a count. The node data structure is:

class listnode
{
  public:
    string s;
    int count;
    listnode *next;
    listnode *prev;
    listnode(): s("") , count(0) , next(NULL) , prev(NULL) {};
    listnode(const string & ss , const int &c): s(ss) , count( c) , next(NULL) , prev(NULL) {};
};
The doubly linked list private data

- Protect data:
  - head: pointer to the head of the list: head->prev == NULL
  - tail: pointer to the tail of the list: tail->next == NULL
  - size: number of nodes in the list

```cpp
class mylist {
    ...
    Private:
        listnode * head;
        listnode * tail;
        int size;
    }
```
mylist public interface

mylist();
¬mylist();
void print();
mylist(const mylist & l);
mylist& operator=(const mylist &l);

void insertfront(const string &s, const int & c);
void insertback(const string &s, const int & c);
void insertbefore(listnode *ptr, const string &s, const int &c);
void insertafter(listnode *ptr, const string &s, const int &c);
void insertpos(const int & pos, const string &s, const int &c);
mylist public interface

void removefront();
void removeback();
void remove(listnode * ptr);
void removepos(const int & pos);

listnode front() const;
listnode back() const;
int length() const;
listnode *search(const string &s);
listnode *findmaxcount();
void removemaxcount();
bool searchandinc (const string &s);
Mylist implementation

- Constructors and destructor
  - Making an empty list (default constructor): head=tail=NULL, size = 0; (See mylist.cpp)
  - Destructor: must use a loop to delete every single node in the list (all nodes are allocated with a new). See mylist.cpp
  - Copy constructor and = operator: Similar logic to destructor: use a loop to walk through each node in the existing list, and insert (just insertback) the same node to the new list.

- The print function (see mylist.cpp)

- The main routines are different versions of insert, remove, and search.
Insert

- Insertback
  - Two cases:
    - Insert to the empty list
    - Insert to list with items.
  - Insert to empty list
    - Create a new node (prev=NULL, next=NULL), both head and tail should point to the new node.

```c
listnode *t = new listnode(s, c);
if (head == NULL) { // list is currently empty, both head and tail
    // should point to the new node
    head = t;
    tail = t;
    size++;
}
```
Insertback

- Insertback to a list with items
  - Step 1: create the new node
    - ListNode *t = new ListNode(s, c)

```
<table>
<thead>
<tr>
<th>NULL</th>
<th>“abc”, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>“whi”, 0</td>
</tr>
<tr>
<td>NULL</td>
<td>“bla”, 0</td>
</tr>
</tbody>
</table>

head

<table>
<thead>
<tr>
<th>NULL</th>
<th>“xxx”, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

tail
```
Insertback

• Insertback to a list with items
  • Step 2: link new node to the tail of the list (next pointer)
    • tail->next = t;

```
head
NULL
"abc", 0
||
"whi", 0
||
"bla", 0
||
tail
NULL
"xxx", 0
NULL
```
Insertback

- Insertback to a list with items
  - Step 3: link new node to the list (prev pointer)
    - \( t->prev = \text{tail}; \)

```
NULL
<table>
<thead>
<tr>
<th>“abc”, 0</th>
</tr>
</thead>
</table>

“whi”, 0

“bla”, 0

“xxx”, 0
| NULL |
```

head

<table>
<thead>
<tr>
<th>head</th>
</tr>
</thead>
</table>

tail

<table>
<thead>
<tr>
<th>tail</th>
</tr>
</thead>
</table>
Insertback

- Insertback to a list with items
- Step 4: tail point to the new node
  - tail = t
- See complete code in mylist.cpp

```
head
  "abc", 0
  "whi", 0
  "bla", 0
  "xxx", 0
  NULL

tail
```

See complete code in mylist.cpp
**Insertbefore**

- Insert before the head is equal to insertfront, which is similar to insertback
- Insertbefore into the middle of the list before ptr
  - A new node is to be added between ptr->prev, and ptr.

![Diagram]

- head
- tail
- ptr
- "abc", 0
- "whi", 0
- "bla", 0
- "xxx", 0
- NULL
Insertbefore

- Insertbefore into the middle of the list before ptr
- A new node is to be added between ptr->prev, and ptr.
- Step 1: create the new node: `listnode* t = new listnode(s, c);`
Insert before

- Insert before into the middle of the list before ptr
  - A new node is to be added between ptr->prev, and ptr.
- Step 1: try to chain the new node to the list
  - t->next = ptr; t->prev = ptr->prev;

```
null
 "abc", 0
 "whi", 0
 "bla", 0
 "xxx", 0
NULL

"yyy", 0
```

ptr
Insertbefore

- Insertbefore into the middle of the list before ptr
  - A new node is to be added between ptr->prev, and ptr.
- Step 2: change ptr->prev’s next pointer
  - ptr->prev->next = t;

```
head
    NULL
     “abc”, 0

     “whi”, 0

     “bla”, 0

    “yyy”, 0

    “xxx”, 0
     NULL
tail
```

 ptr

Insertbefore

- Insertbefore into the middle of the list before ptr
  - A new node is to be added between ptr->prev, and ptr.
- Step 3: change ptr’s prev pointer (see mylist.cpp)
  - \( \text{ptr->prev} = t; \)
Can step 2 and step 3 change order?

\[ \text{ptr} \rightarrow \text{prev} = t; \]
\[ \text{ptr-prev} \rightarrow \text{next} = t; \]
Remove

- Removefront:
  - Two cases:
    - the list has only one element need to make empty list out of it.
      delete head;
      head = tail = NULL;
      size = 0;
      return;
    - The list has more than on elements
Remove

- **Removefront:**
  - The list has more than one elements
  - Step 1: listnode *t = head;

```c
Step 1: listnode *t = head;
```
Remove

- **Removefront:**
  - The list has more than one element
  - Step 2: Advance head: head = head->next;
Remove

- Removefront:
  - The list has more than one element
  - Step 3: delink the prev of head: head->prev = NULL;

```c
head = NULL
	"abc", 0

null
	"whi", 0

null
	"bla", 0

"xxx", 0
null
```
Remove

- **Removefront:**
  - The list has more than one element (see mylist.cpp)
  - Step 4: delete t;
Removemiddle

- Remove an item pointed to by ptr
- Step 1: change ptr->prev’s next pointer

ptr->prev->next = ptr->next;
Removemiddle

- Remove an item pointed to by ptr
- Step 2: change ptr->next’s prev pointer

ptr->next->prev = ptr->prev;
Removemiddle

- Remove an item pointed to by ptr
- Step 3: delete ptr;

```
head
  +------------------
  | NULL             |
  +------------------
  | “abc”, 0         |
  +------------------
  | “whi”, 0         |
  +------------------
  | “xxx”, 0         |
  +------------------
  | NULL             |
  +------------------
```

ptr

```
Search

- Use the while loop to walk through every nodes in the list (see mylist.cpp)

    listnode *t = head;

    while ((t!=NULL) && (t->s != s)) t = t->next;
    return t;