Dynamic Memory - Lists: Handout

This is an outline of the lecture that is intended to provide you with the images and code. It is not a tutorial. Students are expected to pay attention and fill in the details.

• Fundamental dynamic memory structure is: list
  ◦ Simplest list: sequential collection of code objects
  ◦ More complex versions: trees, complex graphs

List: 

![List Diagram]

Tree: 

![Tree Diagram]

• Code objects in list or graph are often called: nodes
class Node {
    public: int item; // data in node
    Node* next; // next node
};

Node* pointer; // pointer to node

pointer = new Node;

- Dynamic data values referenced indirectly; e.g.,

    pointer -> item = 5;
• Lists usually have “head” node different than other nodes; i.e.,

    Node* head;   // points to head of list
- List can be read recursively or with while.

- Idea is to repetitively move down list until NULL encountered in next pointer of node.

```c
void DisplayResults (Node* current) {
    if ( ! (current == NULL) ) {
        cout << current -> item << " ";
        current = current -> next;
        DisplayResults (current); }
}
```
• example: Build a list: build first node (code fragment)

```cpp
// -----------------------------
class Node {
    public: int item;       // data in node
        Node* next;  // points to next node 
};
// -----------------------------
void BuildFirstNode (int & inputValue, Node* & head,
        Node* &current, Node* &pointer) {
    cin >> inputValue;
    pointer = new Node; // new node added to list
    pointer -> item = inputValue; // value inserted
    pointer -> next = NULL; // next pointer to NULL
    head = pointer; // head point to first node
    // set current to point to first node in list
    current = head; }
// -----------------------------
......
// -----------------------------
int inputValue;
Node* pointer; // "pointer" points to Node
Node* head;    // points to head of list
Node* current; // points to where program is
BuildFirstNode (inputValue,head,current,pointer);
```

![Diagram of list construction](image)
• example: Build a list: add cell to a list (code fragment)

```cpp
// ------------------------------------
class Node {
    public: int item;     // data in node
        Node* next;   // points to next node 
};
// ------------------------------------
void BuildFirstNode (int & inputValue, Node* & head,
                      Node* &current, Node* &pointer) {
    cin >> inputValue;
    pointer = new Node; // new node added to list
    pointer -> item = inputValue; // value inserted
    pointer -> next = NULL; // next pointer to NULL
    head = pointer; // head point to first node
    // set current to point to first node in list
    current = head;
 }
// ------------------------------------
......
// ------------------------------------
int inputValue;
Node* pointer; // "pointer" points to Node
Node* head; // points to head of list
Node* current; // points to where program is
BuildFirstNode (inputValue,head,current,pointer);
```
Dynamic Memory - Lists: Handout

• example: Display list (code fragment)

```c++
// Node class definition
class Node {
    public: int item;  // data in node
        Node* next;  // points to next node
};

void DisplayResults (Node* current) {
    if ( ! (current == NULL) ) {
        cout << current->item << " ";
        current = current->next;
        DisplayResults (current); } }

int main () {
    ...  
    Node* head;  // points to head of list
    ...  
    DisplayResults (head);  // display results
}
```

![Diagram of list structure with pointers and nodes]
• When nodes are deleted, process:
  ○ Locate node to be deleted
  ○ Disconnect from list by changing pointers
  ○ Return disconnected node to system

- In middle of list, if current (\texttt{cur}) pointer references node to be deleted, should be pointer indicating previous (\texttt{prev}) node (otherwise previous node lost).

\[
\text{prev} \rightarrow \text{next} = \text{cur} \rightarrow \text{next} \\
\text{or} \hspace{1cm}
\text{temporary} = \text{cur} \rightarrow \text{next} \\
\text{prev} \rightarrow \text{next} = \text{temporary}
\]

- Works fine for all nodes (including last), except for first node (\texttt{prev} does not point to \texttt{head}). Special case:

\[
\text{head} = \text{head} \rightarrow \text{next}
\]
• When lists have new nodes inserted
  ◦ Either node is added at end or beginning of list, or
  ◦ Node is inserted somewhere in between ends; general method:
    ♦ Acquire new data
    ♦ Create new node, store new data in it
    ♦ Determine point of insertion
    ♦ Insert new node into linked list by changing pointers

```
NewPtr = new node;
// insert new data into node here
NewPtr->Next = Cur;
    Prev->Next = NewPtr;
```
- Special Case: Insert at *list front*:

\[
\text{NewPtr} = \textbf{new} \text{ node}; \\
\text{NewPtr} \rightarrow \text{Next} = \text{Head}; \\
\text{Head} = \text{NewPtr};
\]

- Special Case: Insert at *list end*:

\[
\text{NewPtr} = \textbf{new} \text{ node}; \\
\text{NewPtr} \rightarrow \text{Next} = \text{Cur}; \\
\text{Prev} \rightarrow \text{Next} = \text{NewPtr};
\]
• **Dummy Nodes:** To reduce programming complexity caused by adding/deleting nodes at list start/end, “dummy” nodes can be used. Values would be outside of normal range. Thus, insertion/deletion would never affect start/end list nodes.

- **Header Nodes**
  - Used to describe lists
  - Usually different type than rest of list

• **More Complex Linear Lists:** Variety of other linear lists; e.g.,
  - Circular (cyclic):

  ![Circular List Diagram]

  ![More Complex Linear Lists Diagram]

• **More Complex Linear Lists:** Variety of other linear lists; e.g.,
  - Circular (cyclic)
  - Two way