Lecture No.03
Linked Lists

CC-213 Data Structures
Department of Computer Science
University of the Punjab

Slides modified very slightly from the late Dr. Sohail Aslam’s lectures at VU

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.

List Using Linked Memory

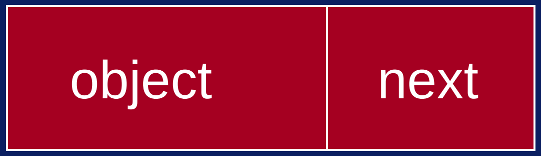
- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.
- Now the first element must *explicitly* tell us where to look for the second element.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.
- Now the first element must *explicitly* tell us where to look for the second element.
- Do this by holding the memory address of the second element

Linked List

- Create a structure called a *Node*.

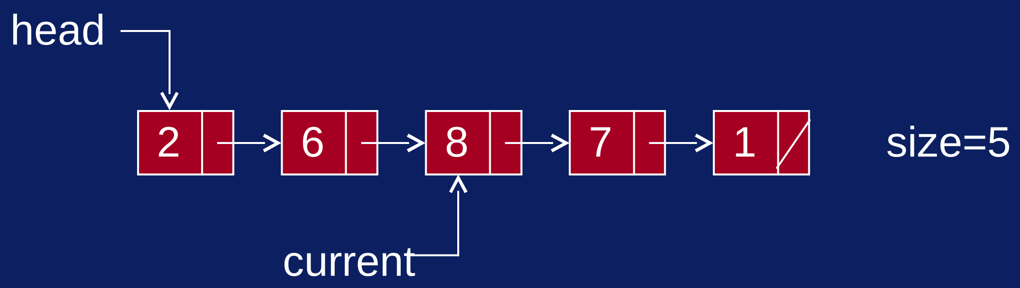


object next

- The *object* field will hold the actual list element.
- The *next* field in the structure will hold the starting location of the next node.
- Chain the nodes together to form a *linked* list.

Linked List

- Picture of our list (2, 6, 7, 8, 1) stored as a linked list:



Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won’t know where the start of the list is.

Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won’t know where the start of the list is.
- The *current* here is a pointer, not an index.

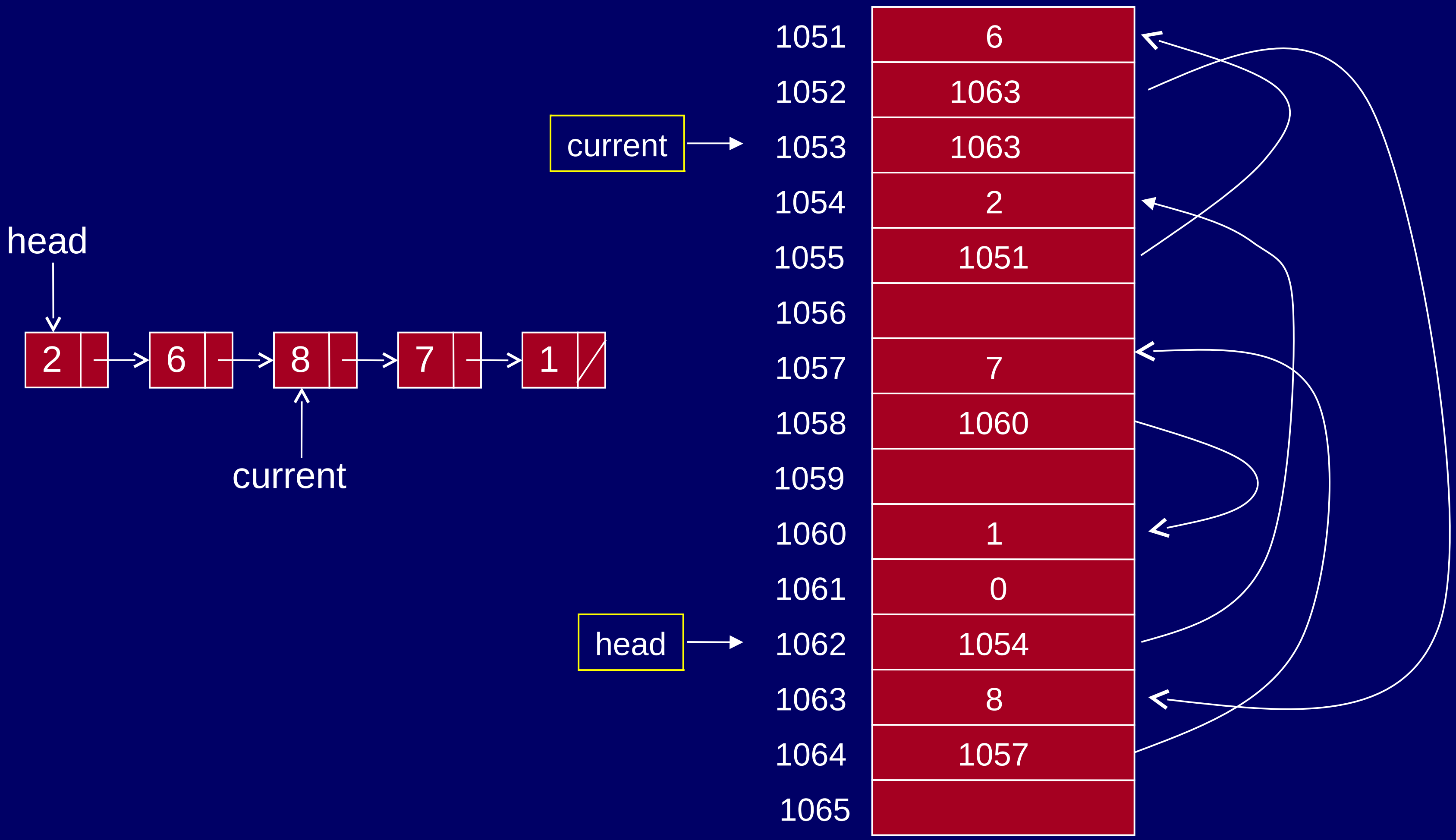
Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won’t know where the start of the list is.
- The *current* here is a pointer, not an index.
- The next field in the last node points to *nothing*. We will place the memory address NULL which is guaranteed to be inaccessible.

Linked List

- Actual picture in memory:

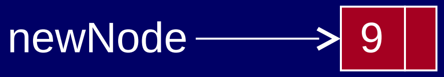


1051	6
1052	1063
1053	1063
1054	2
1055	1051
1056	
1057	7
1058	1060
1059	
1060	1
1061	0
1062	1054
1063	8
1064	1057
1065	

Linked List Operations

- add(9): Create a new node in memory to hold ‘9’

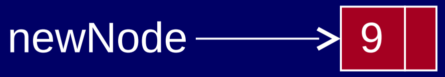
```
Node* newNode = new Node(9);
```



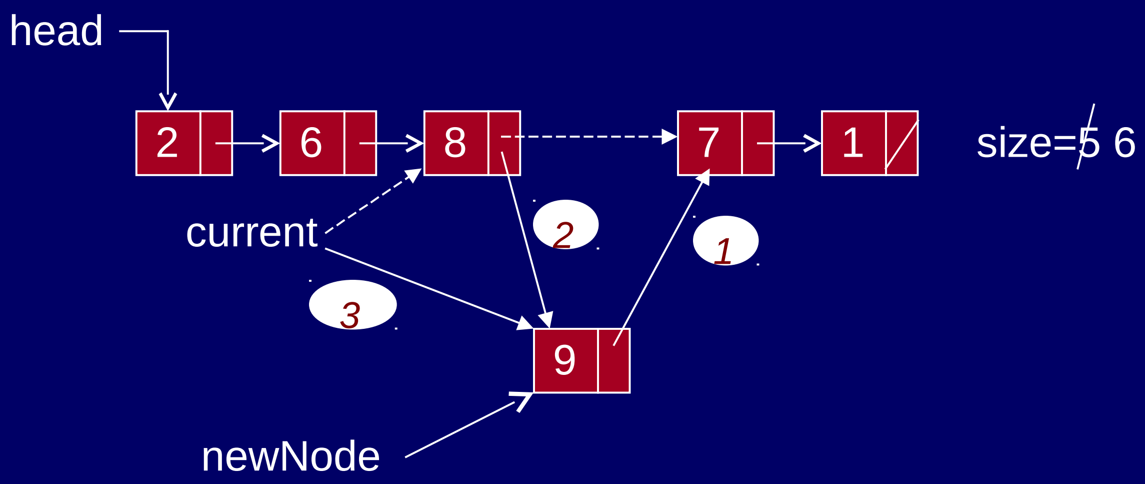
Linked List Operations

- add(9): Create a new node in memory to hold ‘9’

```
Node* newNode = new Node(9);
```



- Link the new node into the list



C++ Code for Linked List

The Node class

```
class Node {
public:
    int get() { return object; };
    void set(int object) { this->object = object; };

    Node *getNext() { return nextNode; };
    void setNext(Node *nextNode)
          { this->nextNode = nextNode; };
private:
    int object;
    Node *nextNode;
};
```

C++ Code for Linked List

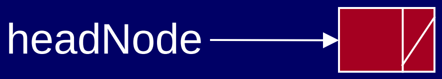
```
#include <stdlib.h>
#include "Node.cpp"

class List {
public:
   // Constructor
   List() {
      headNode = new Node();
      headNode->setNext(NULL);
      currentNode = NULL;
      size = 0;
   };
```

C++ Code for Linked List

```
void add(int addObject) {
  Node* newNode = new Node();
  newNode->set(addObject);
  if( currentNode != NULL ){
         newNode->setNext(currentNode->getNext());
      currentNode->setNext( newNode );
      lastCurrentNode = currentNode;
      currentNode = newNode;
      }
      else {
      newNode->setNext(NULL);
      headNode->setNext(newNode);
      lastCurrentNode = headNode;
         currentNode =  newNode;
      }
      size++;
};
```

Building a Linked List

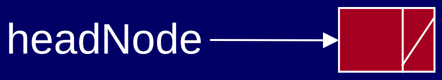
```
List list;
```



size=0

Building a Linked List

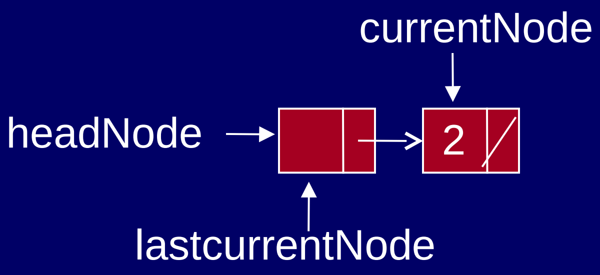
```
List list;
```



headNode

size=0

```
list.add(2);
```



currentNode

headNode

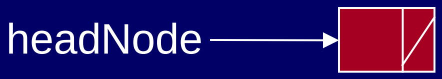
2

lastcurrentNode

size=1

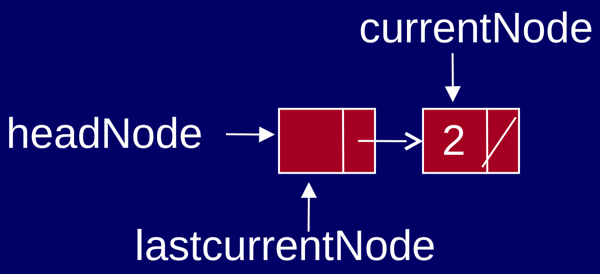
Building a Linked List

```
List list;
```



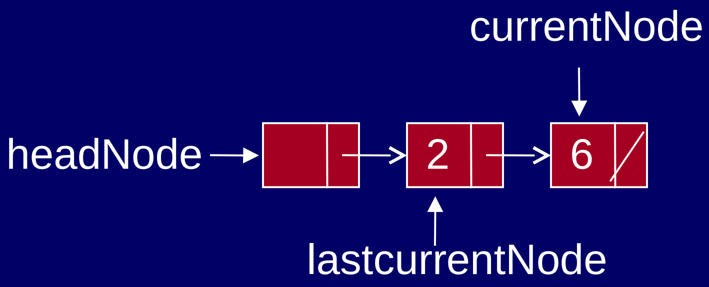
size=0

```
list.add(2);
```



size=1

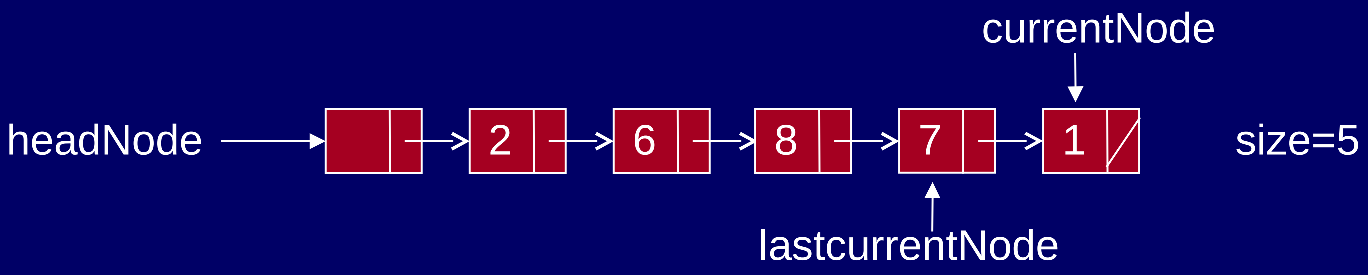
```
list.add(6);
```



size=2

Building a Linked List

```
List.add(8); list.add(7); list.add(1);
```



C++ Code for Linked List

```
int get() {
   if (currentNode != NULL)
       return currentNode->get();
};
```

C++ Code for Linked List

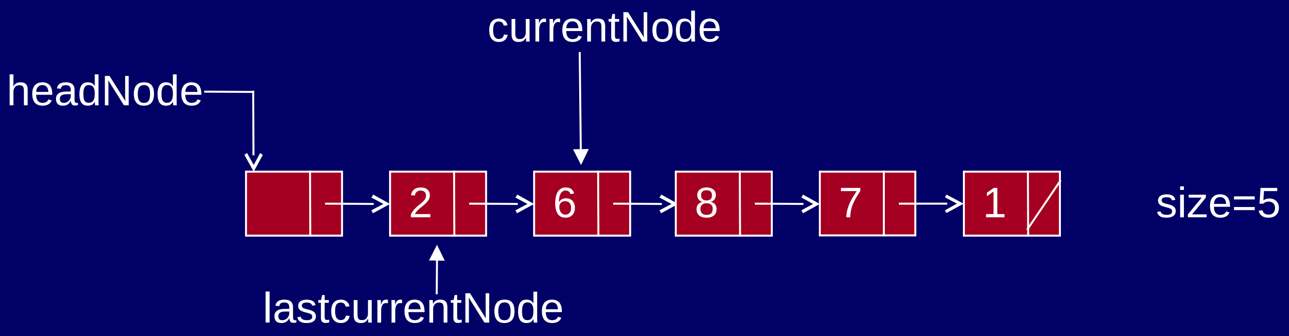
```
bool next() {
   if (currentNode == NULL) return false;

   lastCurrentNode = currentNode;
   currentNode = currentNode->getNext();
   if (currentNode == NULL || size == 0)
      return false;
   else
      return true;
};
```

C++ Code for Linked List

```
// position current before the first
// list element
void start() {
      lastCurrentNode = headNode;
      currentNode = headNode;
};
```

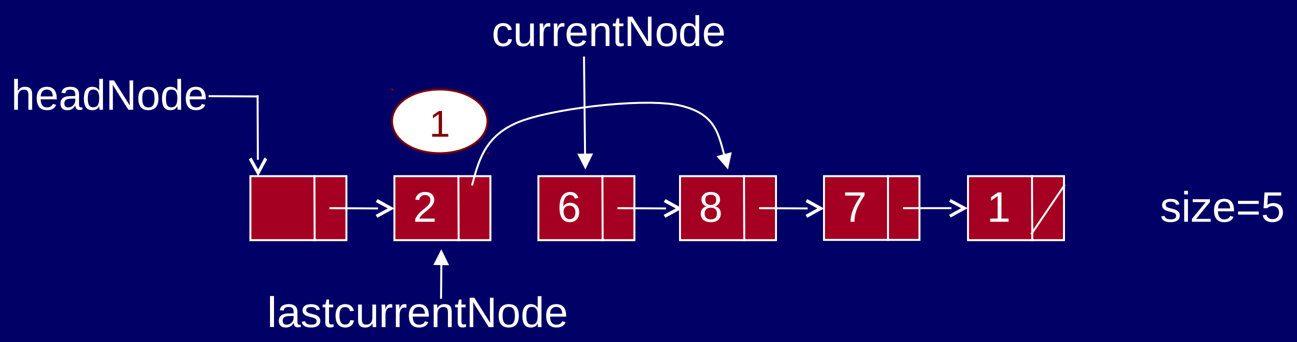
C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
     lastCurrentNode->setNext(currentNode->getNext());
     delete currentNode;
     currentNode = lastCurrentNode->getNext();
     size--;
  }
};
```



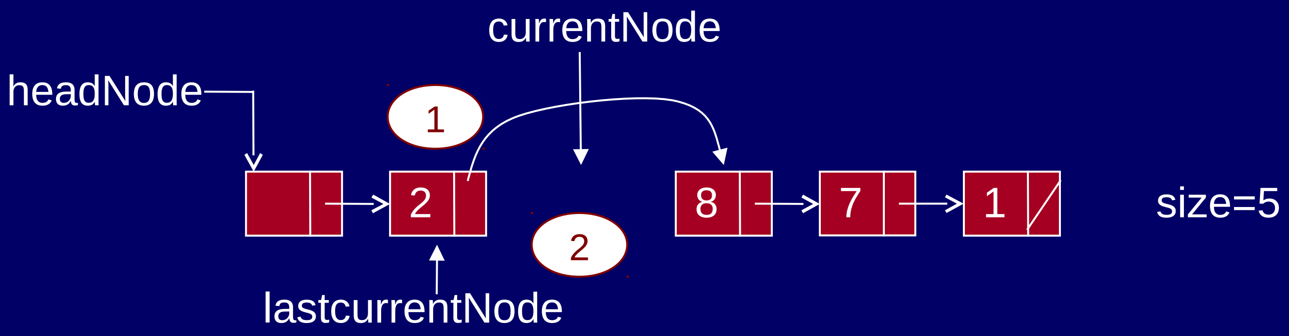
C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    lastCurrentNode->setNext(currentNode->getNext());
    delete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
  }
};
```



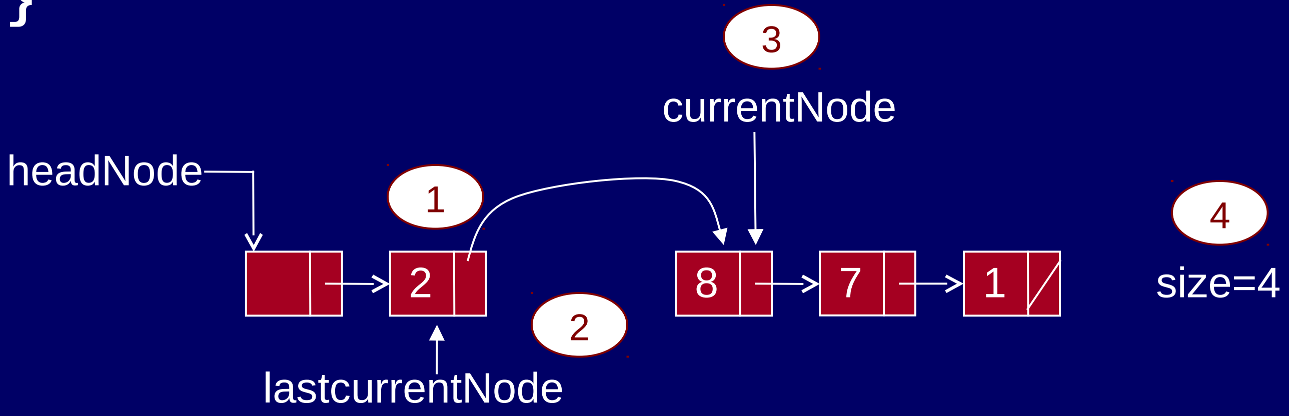
C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    1 stCurrentNode->setNext(currentNode->getNext());
    2 lete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
  }
};
```



C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    1 stCurrentNode->setNext(currentNode->getNext());
    2 lete currentNode;
    3 rrentNode = lastCurrentNode->getNext();
    4 ze--;
  }
};
```



C++ Code for Linked List

```
int length()
{
   return size;
};

private:
   int size;
   Node *headNode;
   Node *currentNode, *lastCurrentNode;
```

Example of List Usage

```
#include <iostream>
#include <stdlib.h>
#include "List.cpp"

int main(int argc, char *argv[])
{
    List list;

    list.add(5); list.add(13); list.add(4);
    list.add(8); list.add(24); list.add(48);
    list.add(12);
    list.start();
    while (list.next())
        cout << "List Element: "<< list.get()<<endl;
}
```

Analysis of Linked List

- add
  - we simply insert the new node after the current node. So add is a one-step operation.

Analysis of Linked List

- add
  - we simply insert the new node after the current node. So add is a one-step operation.
- remove
  - remove is also a one-step operation

Analysis of Linked List

- add
  - we simply insert the new node after the current node. So add is a one-step operation.
- remove
  - remove is also a one-step operation
- find
  - worst-case: may have to search the entire list

Analysis of Linked List

- add
  - we simply insert the new node after the current node. So add is a one-step operation.
- remove
  - remove is also a one-step operation
- find
  - worst-case: may have to search the entire list
- back
  - moving the current pointer back one node requires traversing the list from the start until the node whose next pointer points to current node.