Pros and cons for storing a list in an array

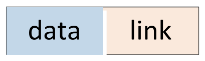
- Support direct (random) access to elements in the list by index.
- Support binary search if the list is sorted.
- Insertion and deletion operation to an ordered list requires O(n) time.
- Size of the array is fixed once it is created. Programmer needs to estimate the required size of the array.
- Handling of array overflow is costly (create a new array and copy the data objects from old array to new array).

Representation of a list as linked list

- Length of a linked list is elastic.
- A linked list can be expanded dynamically.
- A linked list only supports sequential access to elements in the list.
- Insertion or deletion can be done in constant time (provided the point of insertion or deletion is determined).

Linked List

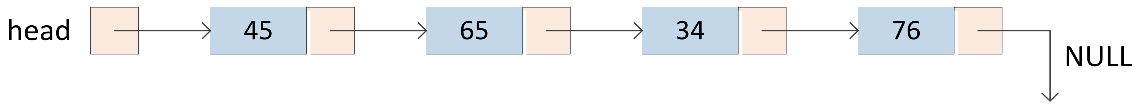
- A linked list is a collection of components called nodes. Nodes are dynamically created or destroyed based on computation requirements.
- A node has two parts, the data (or info) and the link (or next). The link is a pointer pointing to the next node.



- The address of the first node in the list is stored in a pointer variable usually called head, first, or list.

- The null pointer (*nullptr* or *NULL*, physical value 0 in C/C++) is used to denote the end of the list, or not a valid address.

Example: a linked list of 4 integers.



In typical C++ implementations, we define the node using struct and design the functions using template such that the programs can be reused for different data types.

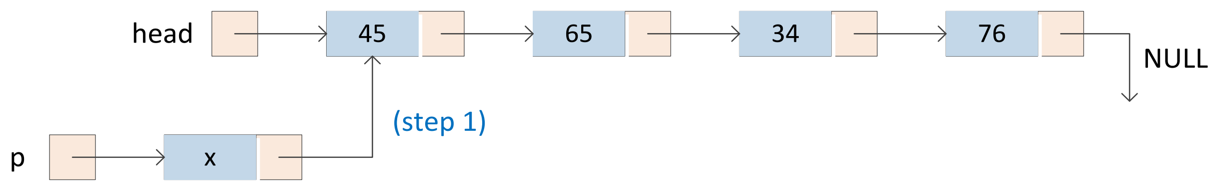
```
template<class Type>
struct node
{
   Type info;
   node<Type> *link; // another commonly used var name is "next"
};

node<Type> *head;
```

Example program codes that operate on a linked list

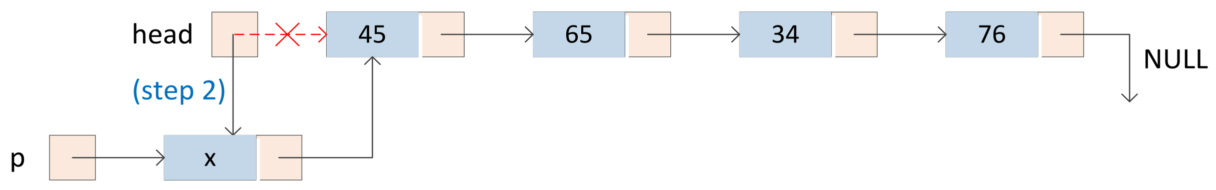
```
// To find the length of the linked list

int len = 0;
node<Type> *p = head;  // traverse the list using p

while (p != nullptr) // p points to a valid node
{
   len++;
   p = p->link; // move to the next node
}
```

```
// Insert a new element x at the front of the list.
// Assume Type = int in the following examples.

node<int> *p = new node<int>; // create the node for storing x
p->info = x;

p->link = head;  // step 1
head = p;        // step 2
```

step 1:



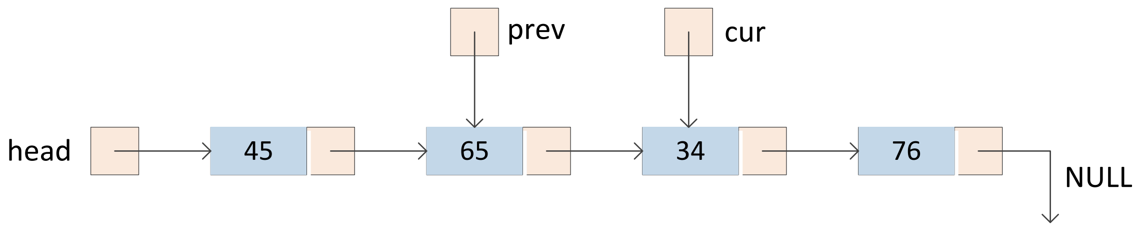
step 2:



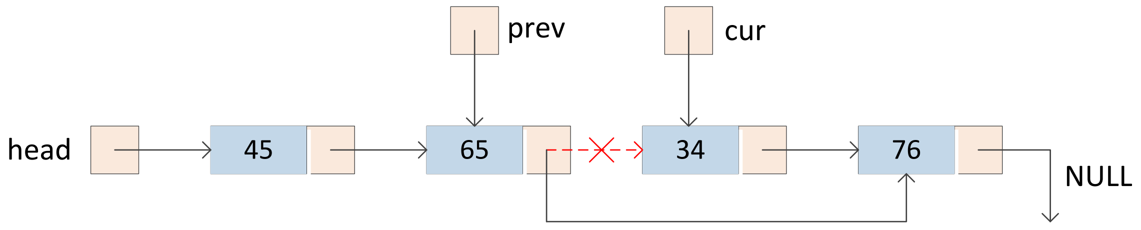
```
// Remove the element x (1st instance) from the linked list

// To remove a node from the linked list, we need to
// know the reference to its predecessor (the previous node).

// Program organization:
// 1. search for the node holding x with a while-loop
// 2. if x is found, remove the node

node<int> *cur = head;
node<int> *prev = nullptr;
// prev points to the predecessor of cur

while (cur != nullptr && cur->info != x) // sequential search
{
   prev = cur;
   cur = cur->link;
}

// if cur == nullptr, x is not found in the linked list

if (cur != nullptr) // cur->info == x
{
   if (prev != nullptr) // cur is not the first node
      prev->link = cur->link;
   else
      head = cur->link;

   delete cur; // free the storage of the removed node
}
```

Diagrams showing the deletion operation:

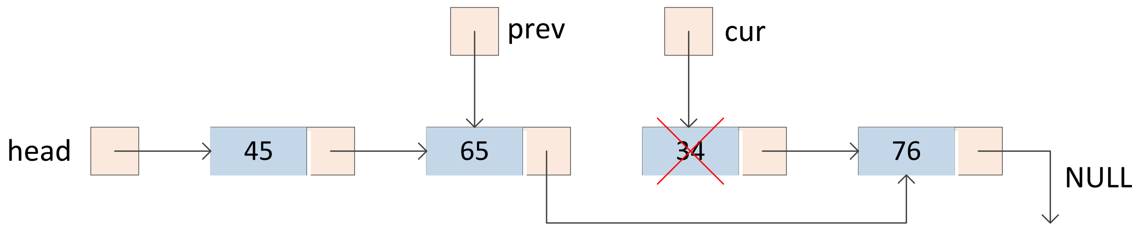
1.Locate the node storing the value x, e.g. x = 34



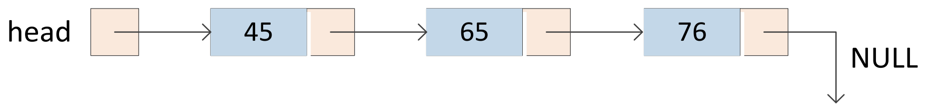
2.Update the links



3.Physically delete the node



Structure of the list after removing the element 34



Implement the operations on linked list as template functions.

```
template<class Type>
int length(const node<Type> *list) // getter function
{                                  // does not modify the list
   // list is a const pointer passed by value
   int len = 0;
   const node<Type> *p = list;

   while (p != nullptr)
   {
      len++;
      p = p->link;
   }
   return len;
}
```

Function to find the min data value in an unordered linked list

```
//Precondition: list is not empty
template<class Type>
Type getMin(const node<Type> *list) // getter function
{
   Type min = list->info;
   const node<Type> *p = list->link;

   while (p != nullptr)
   {
      if (p->info < min)
         min = p->info;

      p = p->link;
   }
   return min;
}
```

Function to remove the node with the min data value in an unordered linked list

```
//Precondition: list is not empty
template<class Type>
Type removeMin(node<Type>*& list) // setter function
{
   // list is passed by reference (may be modified)
   node<Type> min = list;
   node<Type> prev = nullptr;  // prev is predecessor of min

   // Search for the min node and its predecessor

   node<Type> *p = list->link; //comparison starts from 2nd node
   node<Type> *q = list;       //q is predecessor of p

   while (p != nullptr)
   {
      if (p->info < min->info)
      {
         prev = q;
         min = p;
      }

      q = p;
      p = p->link;
   }

   // remove the node min

   Type minValue = min->info;
   if (prev != nullptr)
      prev->link = min->link;
   else  // min is the 1st node
      list = min->link;

   delete min;
   return minValue;
}
```

Function to insert an element to an ordered linked list

```
template<class Type>
void insert(node<Type>*& list, Type e) // setter function
{
   node<Type> *p = new node<Type>;
   p->info = e;

   node<Type> *prev = nullptr;
   node<Type> *cur = list;

   // Determine the point of insertion
   while (cur != nullptr && e > cur->info)
   {
      prev = cur;
      cur = cur->link;
   }

   // Perform the insertion
   if (prev == nullptr)  // insert e at front
   {
      p->link = list;
      list = p;
   }
   else  // insert e after node prev
   {
      p->link = prev->link;
      prev->link = p;
   }
}
```

Remark:

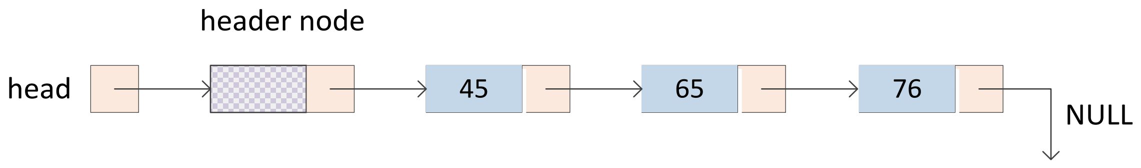
- Null-pointer exception is a common error in programs that manipulate linked list.
- A pointer must be properly initialized or tested for not equal to NULL before you can use it to access a data member or the next node.

Function to merge two ordered linked lists

```
template<class Type>
node<Type>* mergeList(const node<Type > *A, const node<Type> *B)
{
   const node<Type> *p = A; // traverse list A with p
   const node<Type> *q = B; // traverse list B with q
   node<Type> *r, *header;
   r = header = new node<Type>; // r is last node in result list
   // header is a dummy node to simplify the codes in the loop
   while (p != nullptr && q != nullptr)
   {
      r->link = new node<Type>; // assert: r != nullptr
      r = r->link;
      if (p->info <= q->info)
      {
         r->info = p->info;
         p = p->link;
      }
      else
      {
         r->info = q->info;
         q = q->link;
      }
   }
   while (p != nullptr)
   {
      r->link = new node<Type>;
      r = r->link;
      r->info = p->info;
      p = p->link;
   }
   while (q != nullptr)
   {
      r->link = new node<Type>;
      r = r->link;
      r->info = q->info;
      q = q->link;
   }
   r->link = nullptr; // set the link of last node to NULL
   r = header->link;  // r points to 1st data node
   delete header;     // release memory space of header node
   return r;
}
```

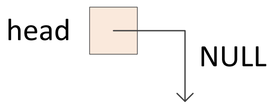
Other variants of linked list

1. Linked list with header node:

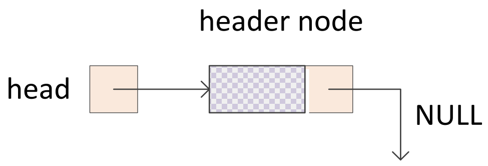


The data field of the header node is not used to store valid data. Some metadata may be stored in the header node (if appropriate).

List does not exist (or not yet created):

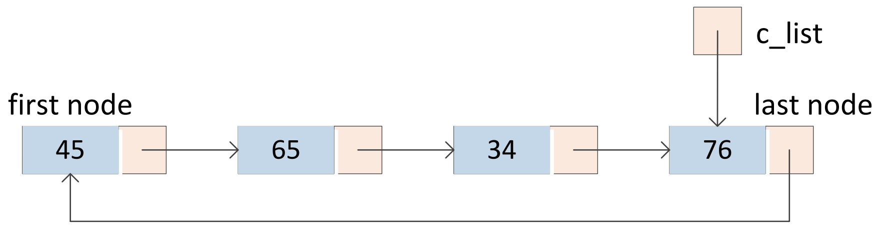


List is empty:



2. Circular list:

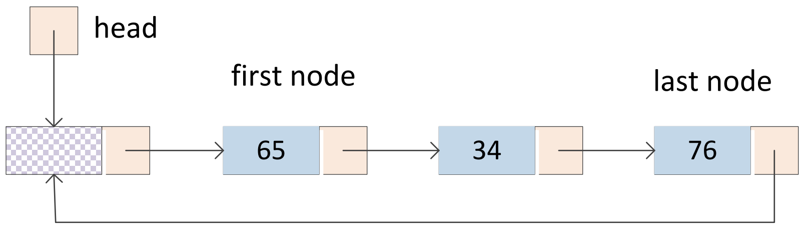
- The `link` of the last node points back to the first node.



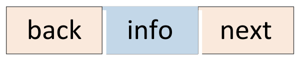
- The pointer variable `c_list` points to the last node in the list.
- We can access the first node in one step:

```
node<Type> *first = c_list->link;
```

3. Circular list with header:



4. Doubly-linked list:

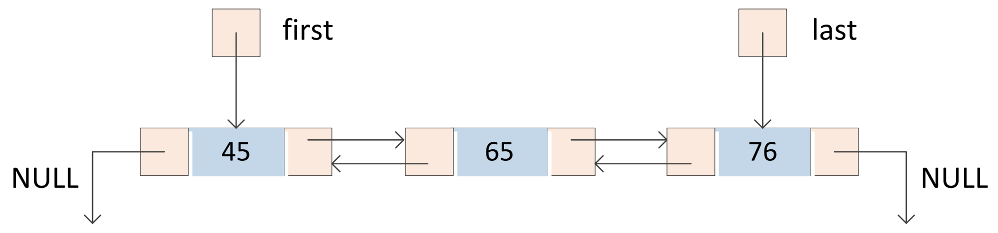


```
template<class Type>
struct DNode
{
   Type info;
   DNode<Type> *next;   //points to successor node
   DNode<Type> *back;   //points to predecessor node
}
```

With a doubly-linked list, we can traverse the list in the forward or backward direction.

It is more convenience to perform insertion and deletion to a doubly-linked list. You don't need to maintain 2 pointer variables.

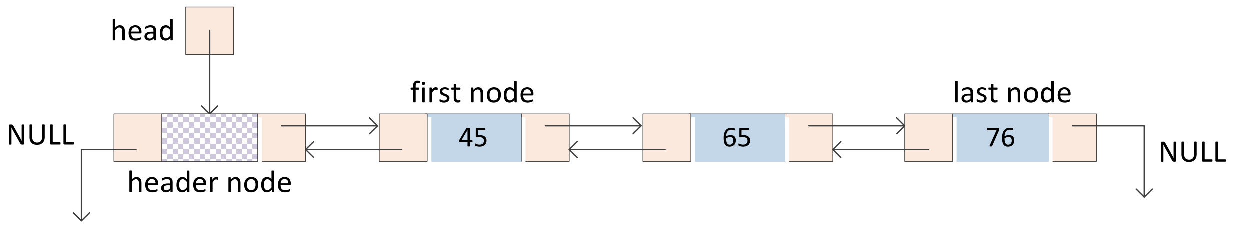
Remark: The `list` container in the C++ standard template library (STL), and the `LinkedList` class in the Java JDK are implemented as doubly-linked list.



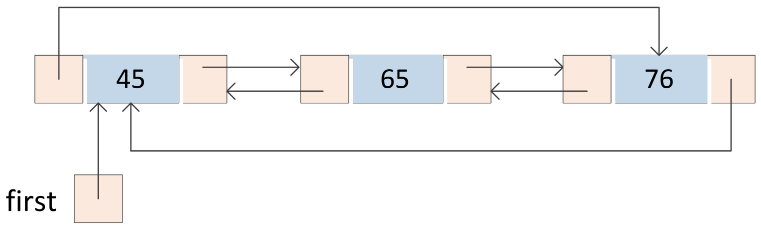
```
//Assume the doubly-linked list is referenced by 2 pointer
//variables first and last

//Insert element x after the current node
DNode<Type> *p = new DNode<Type>;
p->info = x;
p->next = cur->next;   // given cur != NULL
p->back = cur;

if (cur->next != nullptr) // cur has a successor
   cur->next->back = p;
else
   last = p;   // p is the last node after insertion

cur->next = p;
```

```
//Remove node p in a doubly-linked list

if (p->next != nullptr)
   p->next->back = p->back;
else   // p is the last node
{
   last = p->back;
   if (last != nullptr)
      last->next = nullptr;
}

if (p->back != nullptr)
   p->back->next = p->next;
else   // p is the first node
{
   first = p->next;
   if (first != nullptr)
      first->back = nullptr;
}
delete p;
```

5. Doubly-linked list with header node

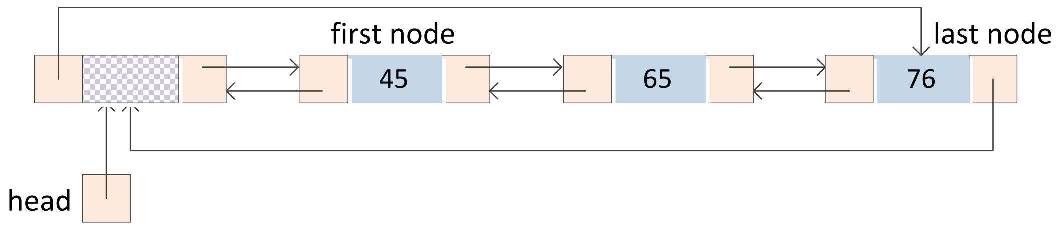


6. Circular doubly-linked list



Circular doubly-linked list is used in the OS for dynamic memory allocation and de-allocation. The algorithm is called boundary-tag method.

7. Circular doubly-linked list with header



The searching problem

Given a set of key-value pairs (where the keys are distinct), we want to find the associated value for a given key.

Time complexity of the search and update operations (insertion or deletion of keys to the data set):

	Time complexity	
Search algorithm	search	insert/delete
sequential search (unordered list)	$O(N)$	$O(1)$
binary search (ordered list maintained in an array)	$O(\log N)$	$O(N)$
binary search tree (balanced)	$O(\log N)$	$O(\log N)$
Hash table (average case and best case)	$O(1)$	$O(1)$

General principle of Hash table

$\Sigma = \{ (K_1, V_1), (K_2, V_2), \ldots, (K_n, V_n) \}$ // data set with *n* records

h(key) is the hash function, and output value of h(key) is between 0 and M-1.
The hash table is an array of size M where $M \geq n$.

Example: assume the keys are unsigned fixed-size integer, and $M = 11$ is a prime number.

h(key) = key % M
$\Sigma = \{ (55, V_1), (41, V_2), (73, V_3), (20, V_4), (35, V_5) \}$

Hash Table

index	bucket (key, value)	
0	$(55, V_1)$	55 % 11 = 0
1		
2	$(35, V_5)$	35 % 11 = 2
3		
4		
5		
6		
7	$(73, V_3)$	73 % 11 = 7
8	$(41, V_2)$	41 % 11 = 8
9	$(20, V_4)$	20 % 11 = 9
10		

To search for a given key x, we use the hash function to compute the home address $home = h(x)$.

- If the *home* bucket is empty, then x does not exist in the set.
- The record stored in the *home* bucket is compared with x. If x matches the key of the record, then the associated value is returned.
- If x does not match the key stored in the *home* bucket, then some additional buckets may be probed depending on the collision resolution strategy used.

Load factor

- The load factor $\lambda = n/M$, where n is the number of elements, and M is the size of the hash table.

Primary clustering

- Let U = domain of the keys.

  M = size of the hash table.

  The hash function h is a mapping h: $U \rightarrow \{0, 1, 2, \ldots, M-1\}$, where $|U| >> M$.
- If this transformation makes certain table locations more likely to occur than others, the chance of collision is increased and the efficiency of searches and insertions is decreased.
- The phenomenon of some table locations being more likely is called primary clustering.
- The preferred hash function spreads the elements uniformly, and does not exhibit primary clustering.

Collision

- A collision is said to have occurred when two distinct keys K_1 and K_2 are hashed into the same bucket, i.e. $h(K_1) = h(K_2)$.

Perfect hash function

- A function h is said to be a perfect hash function with respect to a given set of keys Σ such that $h(K_1) \neq h(K_2)$ for any two distinct keys K_1, $K_2 \in \Sigma$.

Some simple hash functions

Bit-Extraction:

- Merely extract a few scattered bits from an element, putting those bits together to form an address.
- A weakness of extraction is that the resulting location depends only on a small subset of the bits of an element.
- A first principle in the design of hash functions is that the hash location should be a function of every bit of the element.

Compression or folding:

- The key is divided into fixed-length segments, and then add them up as binary numbers or take their exclusive-or.
- For example, keys are character strings, key = "THE",

  h("THE") = 0101 0100 xor 0100 1000 xor 0100 0101 = 0101 1001 = 89_d
- In general, if the keys are character string, then breaking the key into individual chars is not a good idea!

  Permutation of the same set of characters will have the same hash value, e.g. h("THE") = h("HET")

Division:

- $h(x) = x \% M$, where M is a prime number.

Multiplication:

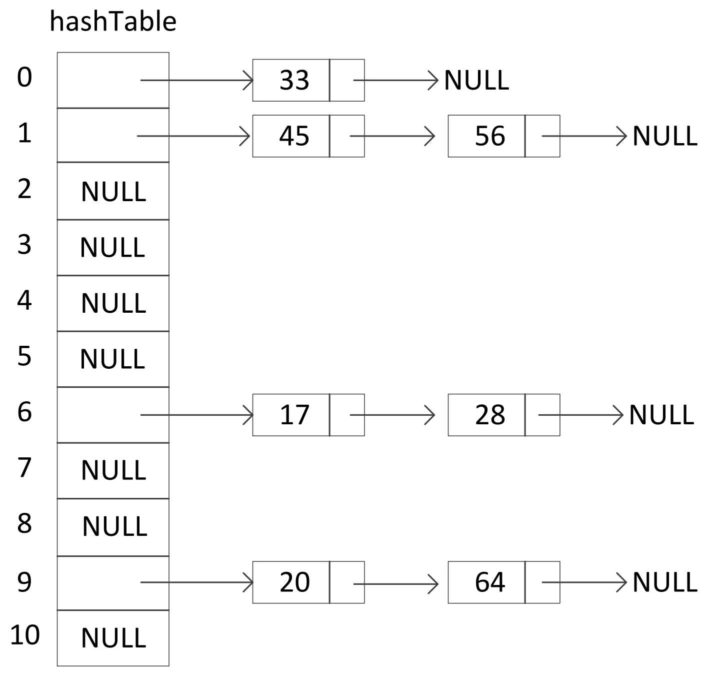
- $h(x) = \lfloor M \times \text{fraction}(\theta \times x) \rfloor$
- where $\theta = (\sqrt{5} - 1) / 2$, or $\theta = 1 - (\sqrt{5} - 1) / 2$

Collision resolution using separate chaining

- The hash table T is organized as an array of linear linked lists, one linked list for each hash bucket.
- To lookup a key, we simply do a sequential search of the given linked list.

  Example:
  Suppose the hash function $h(x) = x \% 11$.
  Let $\Sigma = \{17, 33, 28, 20, 45, 64, 56\}$

  After inserting all the keys, the hash table looks as follows:



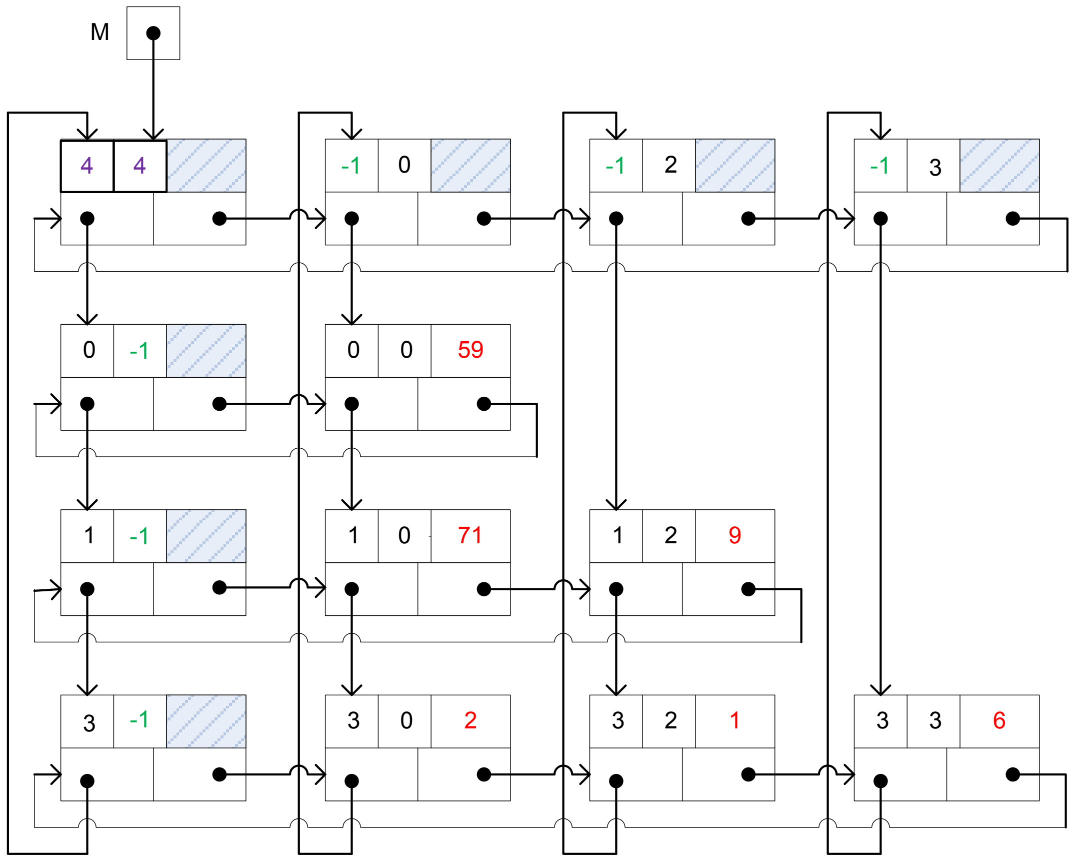
Remarks:
- The hash table class in the Java JDK is implemented using separate chaining.
- Performance of the hash table depends on the load factor. Usually, the load factor is maintained to be no more than 0.75 (threshold chosen by the Java designer).

Orthogonal list: Linked representation of sparse matrix

Non-zero elements in a row (or column) are connected in a circular linked list with header node.

```
// In this example, struct node is NOT defined as template
struct node {  int row, col;
               double value;
               node *down, *right;
            };

node *M;  //reference pointer to the sparse matrix
```

Example: $M = \begin{bmatrix} 59 & 0 & 0 & 0 \\ 71 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 6 \end{bmatrix}$



- In this example, values of `nRow` and `nCol` of the sparse matrix are stored in the dummy header node pointed at by `M`.
- The header of each circular list stores the respective row/column number.

```
//Return the value of matrix[i][j]

double getValue(node *M, int i, int j)
{
   node *p, *q;
   int foundRow = 0;

   p = M->down;
   while (p != M && !foundRow)
   {
      if (p->row < i)
         p = p->down;
      else if (p->row == i) //found the row
         foundRow = 1;
      else
         p = M; //terminate the loop
   }

   if (foundRow) //p points to header of row i
   {
      q = p->right;
      while (q != p)
      {
         if (q->col < j)
            q = q->right;
         else if (q->col == j)
            return q->value;  //value of matrix[i][j]
         else
            q = p; //terminate the loop
      }
   }

   return 0; //matrix[i][j] == 0
}
```

Implementing linked list as a class in C++

Operations that we would perform on a linked list object:

- Initialize the list.
- Clear the list.
- Determine if the list is empty.
- Print the list.
- Find the length of the list.
- Make a copy of the list, e.g. assignment operator= and the copy constructor.
- Search the list for a given item.
- Insert an item to the list.
    - The requirement of the insert operation depends on the representation invariant and the intended uses of the list.
    - For ordered list, we need to maintain the ordering of elements in the list.
    - If it is used as a queue, insertion is performed at the rear (end of list).
    - If it is used as a stack, insertion is performed at the front.
- Remove an item from the list. Similar to the case of insertion.
    - If the list is used as a queue or stack, removal is performed at the front.
- Traverse the list (in the application program that uses the linked list object), i.e. retrieve the elements one by one (in some specific order) to carry out the required computation on each node.
    - The user of the linkedList class does not have access to the protected member variables. Hence, the linkedList class provides an iterator to allow the user to carry out the traversal.
    - An iterator is an object that produces each element of a container, one at a time.
    - The two basic operations on an iterator are the dereference operator *, and the pre-increment operator ++ (advance to the next element). Refer to p.22 for an example on the use of the iterator.
- There can be other operations on the linked list, e.g. reverse the list, merge two lists, etc.

We want the linked list class to be generic such that it can be used to hold objects of different data types.

```
// file: linkedList.h

#ifndef LINKED_LIST_H
#define LINKED_LIST_H
#include <ostream>  // use cout in the print() function

template<class Type>  //definition of node
struct node
{
   Type info;
   node<Type> *link;
};

template<class Type>     //The functions are simple, hence,
class linkedListIterator //they are included in the class definition
{
private:
   node<Type> *cur;

public:
   linkedListIterator() {  cur = nullptr;  }

   linkedListIterator(node<Type> *ptr) {  cur = ptr;   }

   Type operator*() {  return cur->info;  }

   linkedListIterator<Type> operator++()  //pre-increment operator
   {
      cur = cur->link;
      return *this;
   }

   bool operator==(const linkedListIterator<Type>& other)
   {  return cur == other.cur;  }

   bool operator!=(const linkedListIterator<Type>& other)
   {  return cur != other.cur;  }

   //Remark: in the Java language, the ListIterator class has its own
   //version of insert and remove methods.
};
```

```
template<class Type>
class linkedList
{
protected:
   int count;            // no. of elements in the list
   node<Type> *first;   // pointer to the first node
   node<Type> *last;    // pointer to the last node

public:
   linkedList(); //default constructor

   linkedList(const linkedList<Type>& other);
   //copy constructor,
   //it is used when an object is passed to a function by value

   const linkedList<Type>& operator=
                              (const linkedList<Type>& other);

   ~linkedList(); //destructor

   void initializeList();
   bool isEmpty() const;
   void print() const;
   int length() const;
   void destroyList(); //clear the list
   virtual bool search(const Type& x) const;
   virtual void insert(const Type& x);
   virtual void remove(const Type& x);

   Type remove_front(); //Remove front node and return the data value.
                        //The list must be non-empty.

   linkedListIterator<Type> begin();
   linkedListIterator<Type> end();

   //There can be additional member functions, but to simplify
   //the discussion, we will only consider the above functions.

private:
   void copyList(const linkedList<Type>& other);
   //private function used in the copy constructor and operator=
};
```

Remark:
The constructor/mutator methods should maintain the consistency of the member variables, i.e. `count > 0`, iff `first != nullptr and last != nullptr`.

```
// Implementations of the member functions

template<class Type>
linkedList<Type>::linkedList()
{
   count = 0;
   first = last = nullptr;
}


template<class Type>
void linkedList<Type>::destroyList()
{  //clear the list, free the storage occupied by the nodes

   node<Type> *temp;

   while (first != nullptr)
   {
      temp = first;
      first = first->link;
      delete temp;
   }
   count = 0;
   last = nullptr;  //first == nullptr, guaranteed by the while-loop
}


template<class Type>
linkedList<Type>::~linkedList()
{
   destroyList();
}

template<class Type>
void linkedList<Type>::copyList(const linkedList<Type>& other)
{
   if (first != nullptr)
      destroyList(); //clear the old contents of the list

   if (other.first == nullptr) // the other list is empty
      return;

   //copy the first node
   first = last = new node<Type>;
   first->info = other.first->info;
   first->link = nullptr;
   //copy the remaining nodes
   node<Type> *p = other.first->link;

   while (p != nullptr)
   {
      last->link = new node<Type>;
      last = last->link;
      last->info = p->info;
      p = p->link;
   }
   last->link = nullptr;
   count = other.count;
}


template<class Type>
void linkedList<Type>::initializeList()
{
   count = 0;
   first = last = nullptr;
}


template<class Type>
linkedList<Type>::linkedList(const linkedList<Type>& other)
{
   initializeList();
   copyList(other);
}


template<class Type>
const linkedList<Type>& linkedList<Type>::operator=
                                (const linkedList<Type>& other)
{
   if (this != &other) //avoid self-copy
      copyList(other);

   return *this;
}

template<class Type>
bool linkedList<Type>::isEmpty() const
{
   return count == 0; // or return first == nullptr;
}
```

```
template<class Type>
void linkedList<Type>::print() const
{
   node<Type> *p;
   p = first;
   while (p != nullptr)
   {
      cout << p->info << " ";
      p = p->link;
   }
   cout << endl;
}


template<class Type>
int linkedList<Type>::length() const
{
   return count;
}


template<class Type>
bool linkedList<Type>::search(const Type& x) const
{
   node<Type> *p;

   p = first;
   while (p != nullptr && p->info != x)
      p = p->link;

   return p != nullptr;
}
```

```
template<class Type>
void linkedList<Type>::insert(const Type& x)
{  //append at the end of the list

   node<Type> *p = new node<Type>;
   p->info = x;
   p->link = nullptr;

   if (first == nullptr)
      first = last = p;
   else
   {  last->link = p;
      last = p;
   }
   count++;
}


template<class Type>
void linkedList<Type>::remove(const Type& x)
{  //remove the first instance of x in the list

   node<Type> *p, *q;  // q is the predecessor of p
   p = first;
   q = nullptr;

   while (p != nullptr && p->info != x)
   {
      q = p;
      p = p->link;
   }

   if (p != nullptr) // p->info == x
   {
      if (p == first)
         first = first->link;
      else
         q->link = p->link;

      if (p == last)
         last = q;

      delete p;
      count--;
   }
}
```

```
template<class Type>
Type linkedList<Type>::remove_front()
{  //Remove the front node and return the data value.
   //Precondition: the list must be non-empty.

   node<Type> *p;
   Type x;

   p = first;
   x = first->info;  //Null-pointer exception occurs
                     //if the list is empty.

   if (first == last)
      first = last = nullptr;
   else
      first = first->link;

   delete p;
   count--;

   return x;
}


template<class Type>
linkedListIterator<Type> linkedList<Type>::begin()
{
   linkedListIterator<Type> temp(first);
   //create an iterator that references the beginning of the list

   return temp;
}


template<class Type>
linkedListIterator<Type> linkedList<Type>::end()
{
   linkedListIterator<Type> temp(nullptr);
   return temp;
}


#endif  //end of the file linkedList.h
```

Example codes that uses a linked list

```
int main()
{
   linkedList<int> list;
   ifstream inFile("testData.txt");
   if (!inFile.is_open())
   {
      cout << "Error: cannot open data file" << endl;
      exit(0); //terminate the program
   }

   while (!inFile.eof()) //not end of file
   {
      int i;
      inFile >> i;    //read in an integer

      if (!inFile.fail())
         list.insert(i); //insert into the linked list
      else
         break;
   }

   inFile.close();
   cout << "Contents of the list: ";
   list.print();

   // codes to compute the sum of the numbers in the list
   linkedListIterator<int> iter = list.begin();
   linkedListIterator<int> eol = list.end();  //end of list

   // Do not have access to the private variables first and last
   // outside the class, hence we traverse the list using the
   // iterator
   int sum = 0;
   while (iter != eol)
   {
      sum += *iter; //get the data value via the iterator
      ++iter;       //advance the iterator to the next item
   }

   cout << "Sum of the list = " << sum << endl;
}
```

In the above implementation of the linked list, the elements in the list are arranged in chronological order (ordered by the insertion time).

If we want the elements to be ordered by the data values, we can implement an ordered list using inheritance.

```
// file: orderedLinkedList.h

#ifndef ORDERED_LINKED_LIST_H
#define ORDERED_LINKED_LIST_H

#include "linkedList.h"

template<class Type>
class orderedLinkedList: public linkedList<Type>
{
   // Inherit the features linkedList class.
   // Only need to redefine the search() and insert() functions
public:
   bool search(const Type& x) const;
   void insert(const Type& x);

};


//Implementations of the member functions

template<class Type>
bool orderedLinkedList<Type>::search(const Type& x) const
{
   node<Type> *p;

   p = first;
   while (p != nullptr && p->info < x)
         p = p->link;

   return (p != nullptr && p->info == x);
}
```

```
template<class Type>
void orderedLinkedList<Type>::insert(const Type& x)
{  //maintain the ordering of the elements in the list

   node<Type> *p = new node<Type>;
   p->info = x;
   p->link = nullptr;

   if (first == nullptr)
      first = last = p;
   else if (x <= first->info)
   {
      p->link = first;  //insert at front
      first = p;
   }
   else //first != nullptr && x > first->info
   {
      node<Type> *prev = first;
      node<Type> *cur = first->link;

      while (cur != nullptr && x > cur->info)
      {
         prev = cur;
         cur = cur->link;
      }

      // insert node p after node prev
      p->link = prev->link;
      prev->link = p;

      if (prev == last)
         last = p;
   }
   count++;
}

#endif  //end of the file orderedLinkedList.h
```