InsertHead LL

```
template <class T>
void List<T >:: ins ert Head(T
n)
{
ListNo de< T> *aNewNode = new
ListNo de< T>(n);
aNewNo de- >_next = _head;
_head = aNewNode;
_size++;
};
```

Remove Head LL

```
template <class T>
void List<T >:: rem ove Head()
{
     if (_size > 0) {
            Lis tNo de< T> *temp
= _head;
            _head = _head- -
>_next;
            delete temp;
            _si ze--;
       }
}
```

Iterative Print LL

```
template <class T>
void List<T >:: pri nt(bool
withNL) {
ListNo de< T> *temp = _head;
      while (temp) {
               cout << temp-> -
_item;
               if (withNL)
                   cout << endl;
               else
                   cout << " ";
               temp = temp-> -
_next;
         }
         cout << endl;
}
```

Exist LL

```
template <class T>
bool List<T>::exist(T n)
{
for (ListNode<T>* ptr = _head; ptr; ptr = ptr-
> return max;
}
return T();
}
```

ReverseOP LL

```
template <class T>
void List<T >:: rev ers eOp() {
ListNo de< T> *previous = NULL;
//start from NULL
ListNo de< T> *current = _head;
ListNo de< T> *next;
while (current != NULL) {
next = curren t-> _next;
curren t-> _next = previous;
previous = current;
current = next;
}
_head = previous;
}
```

Extract Max LL

```
template <class T>
T List<T >:: ext rac tMax()
{
if (_head != NULL) {
ListNo de< T>* largest = _head;
ListNo de< T>* location = _head;
T max = larges t-> _item; //
//allo cating the node before
the largest.
for (ListN ode <T>* ptr = _head;
ptr; ptr = ptr->_ next) {
if (ptr-> _next != NULL) {
if (ptr-> _ne xt- >_item >
larges t-> _item) {
largest = ptr->_ next;
location = ptr;
max = larges t-> _item;
}
}
}
// removing and joining
if (largest == _head) {
_head = larges t-> _next;
delete largest;
_size--;
}
else {
locati on- >_next = larges t-> -
_next;
delete largest;
_size--;
}
```

Extract Max LL (cont)

```
> return max;
}
return T();
}
```

Operator Overloading

```
#include <iostream>
         using namespace std;
         class Cal {
                  public:
         static int add(int a,int
b){
                          return a
+ b;
                }
         static int add(int a,
int b, int c)
                {
                          return a
+ b + c;
                }
        };
        int main(void) {
                 Cal C; // class
object declar ation.
                  cou t<< C.a -
dd(10, 20)<<endl;
                  cou t<< C.a -
dd(12, 20, 23);
                return 0;
        }
```

Hash Insert

```
hash-insert(key, data)
int i = 1; // num of collisions
while (i <= m) { // Try every bucket
int bucket = h(key, i);
if (T[bucket] == null){ // Found an empty
bucket
T[bucket] = {key, data}; // Insert key/data
return success; // Return
}
i++;
}
HandeError(); // Table full!
```

Hash Search

```
>_next) {
if (ptr->_item == n)
{return true;}
}
return false;
}
```

```
hash-search(key)
int i = 1;
while (i <= m) {
int bucket = h(key, i);
if (T[bucket] == null) // Empty bucket!
return key-not-found;
if (T[bucket].key == key) // Full bucket.
return T[bucket].data;
i++;
}
return key-not-found; // Exhausted entire
table
```

Quick Sort

```
int partition (int arr[], int
low, int high)
{
	int pivot = arr[high]; //
pivot
	int i = (low - 1); //
Index of smaller element

	for (int j = low; j <=
high - 1; j++)
	{
		// If current
element is smaller than or
		// equal to pivot
		if (arr[j] <=
pivot)
		{
			i++; //
increment index of smaller
element
```

By **BearTeddy**
cheatography.com/bearteddy/

Published 21st June, 2019.
Last updated 21st June, 2019.

Quick Sort (cont)

```
>            swap(&arr[i], &arr[j]);
       }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

/* The main function that implements
QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
      at right place */
      int pi = partition(arr, low, high);

      // Separately sort elements before
      // partition and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
```