- O(1): Constant time
- $O(\log_2 n)$: Logarithmic time
- O(n): Linear time
- $O(n\log_2 n)$: Log-linear time
- $O(n^2)$: Quadratic time
- $O(n^3)$: Cubic time
- $O(n^k)$: Polynomial time
- $O(2^n)$: Exponential time

```
// Function to perform insertion sort
void insertionSort(int arr[], int n){

   for (int i = 1; i < n; i++)
   {
      int key = arr[i];
      int j = i - 1;

      while (j >= 0 && arr[j] > key)
      {
         arr[j + 1] = arr[j];
         j--;
      }

      arr[j + 1] = key;
   }
}
```

```
//Stack Reverse Order
#include <iostream>
#include <stack>
using namespace std;
int main() {
stack<int> s;
s.push(10);
s.push(20);
s.push(30);
while(!s.empty()) {
cout << s.top() << " "; // output: 30 20 10
s.pop(); // remove the top item
}
}
```

```
//Josephus
#include <iostream>
using namespace std;
struct node {
int info; // data
node* link;
};
node* head = NULL;
node* last = NULL;
void insert(int x) {
node* ptr = new node;
ptr->info = x;
ptr->link = head;
if (head == NULL) {
head = ptr;
}else{
last->link = ptr;
}
last = ptr; //point to head
}
void remove(node*& head, int k) {
for (int i = 1; i < k; i++) {
head = head->link;
}
node* cur = head->link->link;
delete head->link;
head = head->link = cur; //linked back two nodes
}
int main() {
int n, k, p=1;
cout << "Please input n: \n";
cin >> n;
cout << "Please input k: \n";
cin >> k;
for (int x = 1; x <= n; x++) {
insert(x);
}
while(p != n) {
p++;
remove(head, k);
}
cout << "The last node is " << head->info << endl;
}
```

```
struct node {
int info; // data
node* link;
};
node* head; // head pointer pointing
to first node
```

```
//Find Length
int len = 0;
node *cur = head; //traverse the list using cur
while (cur != NULL) { // cur points to a valid node
len++;
cur = cur->link; //move to the next node
}
```

```
//Search an element x from the beginning of the list
node *cur = head;
while (cur != NULL && cur->info != x) { // search x
cur = cur->link;
}
// cur now points to target x or is NULL (x not exist)
```

```
// Insert a new element x at the front of the list
node *p = new node; // create the node dynamically for
storing x
p->info = x;
p->link = head; // step 1
head = p; // step 2
```

```
// Simplified version: Insert a new element x into an ordered list
node *p = new node;
p->info = x;
node *prev = head; // point to the dummy header
node *cur = head->link; // point to 1st node
while (cur != NULL && x > cur->info) { // search position
prev = cur;
cur = cur->link;
}
p->link = prev->link;
prev->link = p;
```

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

```
// singly linked list with header node
void removeLastNode(node *list) {
   node *p = list;
   //assert: non-empty list
   if (p->link == NULL) return;

   //non-empty => p->link != NULL
   while (p->link->link != NULL)
      p = p->link;
   delete (p->link);
   p->link = NULL;
}
```

```
//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.
node *cur = head;
node *prev = NULL; // prev points to the predecessor of cur
while (cur != NULL && cur->info != x) { // search x
prev = cur;
cur = cur->link;
}
// if cur == NULL, x is not found in the linked list
if (cur != NULL) { // cur->info == x
if (prev != NULL) // why checking this?
prev->link = cur->link; // skip cur node
else // cur is the first node in the list
head = cur->link; // x is the first node
delete cur; //free the storage of the removed node
}
```

```
//Insert a new element x into an ordered list
node *p = new node;
p->info = x;
if (head == NULL || x <= head->info) {
p->link = head; //insert at front
head = p;
} else { //head != NULL && x > head->info
node *prev = head; // x to be inserted between
prev and cur
node *cur = head->link; // i.e. prev->info < x <=
cur->info
while (cur != NULL && x > cur->info) { //
search position
prev = cur;
cur = cur->link;
}
// end-of-list OR x <= cur->info, so insert node
p after node prev
p->link = prev->link;
prev->link = p;
}
```

```
// Output: a pointer p points to the last node of
the list
node* searchLastNode(node *list) {
   node *p;
   p = list->link;
   if(p == NULL)                    // empty
list
      return p;
   while (p->link != NULL)          // reach
the end
      p = p->link;
   return p;
}
```