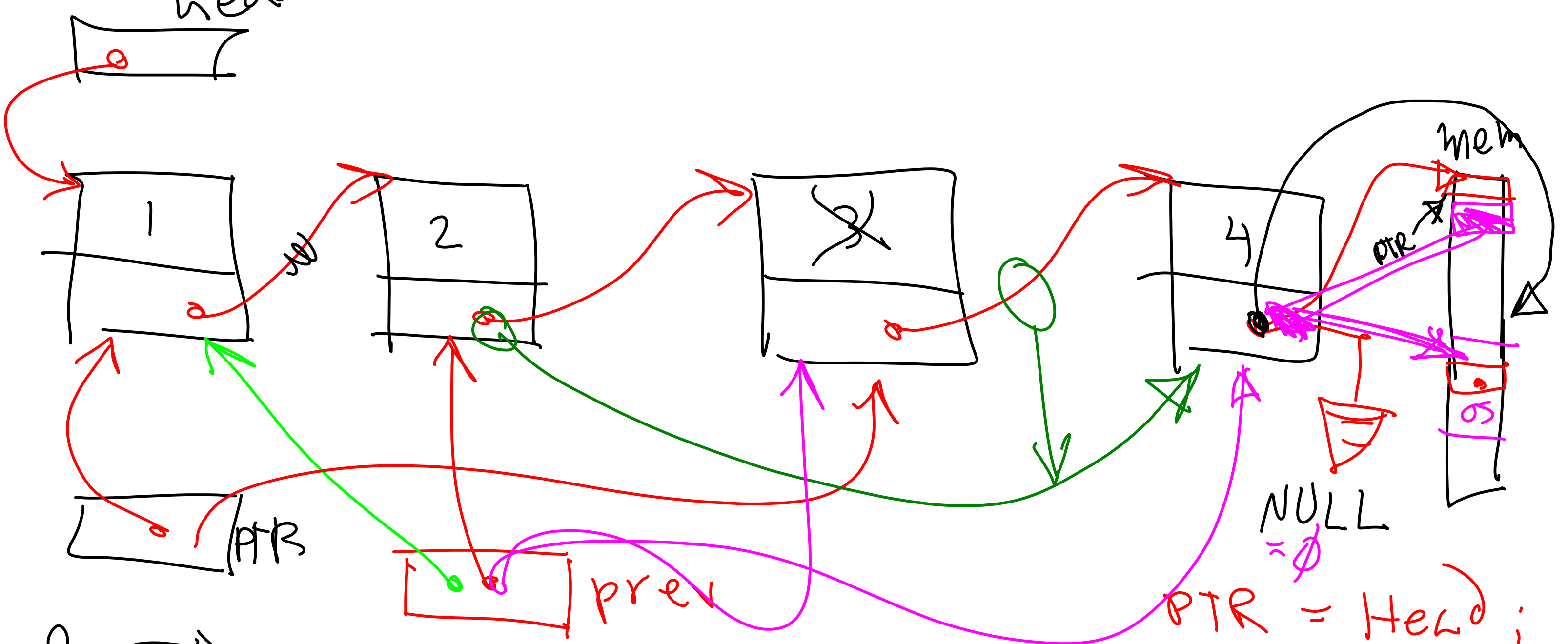
```
class Node {
  public:
      Node( int ); //--constructor
      int x;
      Node *next;
};
```

```
Node::Node( int val ): {
    x = val;
}
```

```
int main(){

  Node *head;
  Node *ptr;
  Node *prev;

  head = new Node(0);    //-------- make a list
  prev = head;
  for( int i=1; i<=10; i++){
      ptr          = new Node( i );
      prev->next = ptr;
      prev        = ptr;
  }
  ptr->next = NULL;

  int i = 0;     //------- print the list
  ptr = head;
  while( ptr != NULL ){
        cout << "\n";
        cout <<    "node "<< i++ << " is " << ptr->x;
        cout << "\n";
        ptr = ptr->next;
  }

  return 0;
}
```

HW



PTR = Head;

prev = Head;

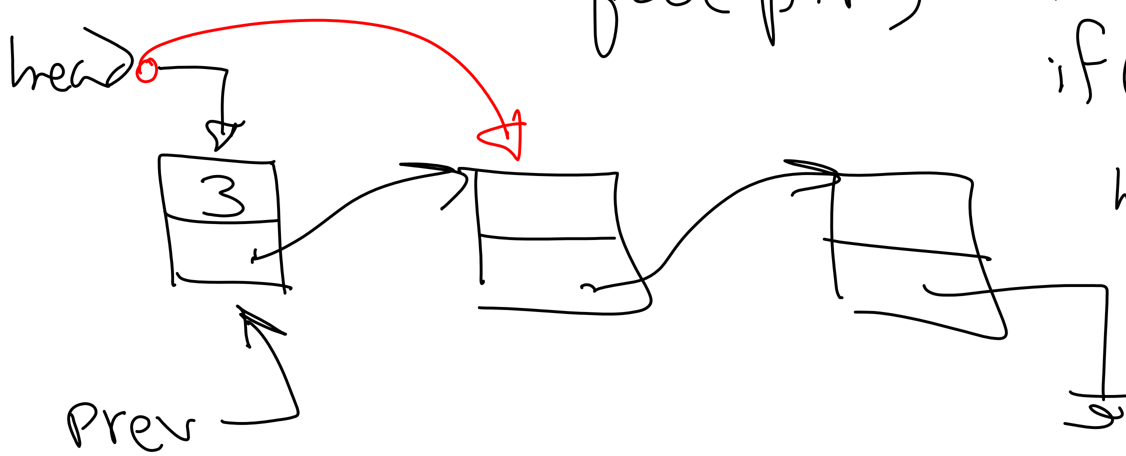
loop

```
if (prev->next->x == 3)
    break;
prev = prev->next
```

```
if (prev->next == NULL)
```

```
prev->next = ptr->next
```

ptr



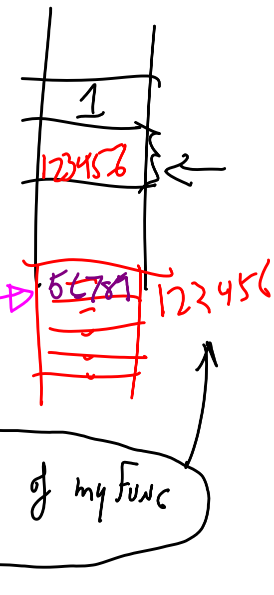

```
free(ptr)
```

```
ptr = head;
if (prev->x == 3)
    head = prev->next
    return
```

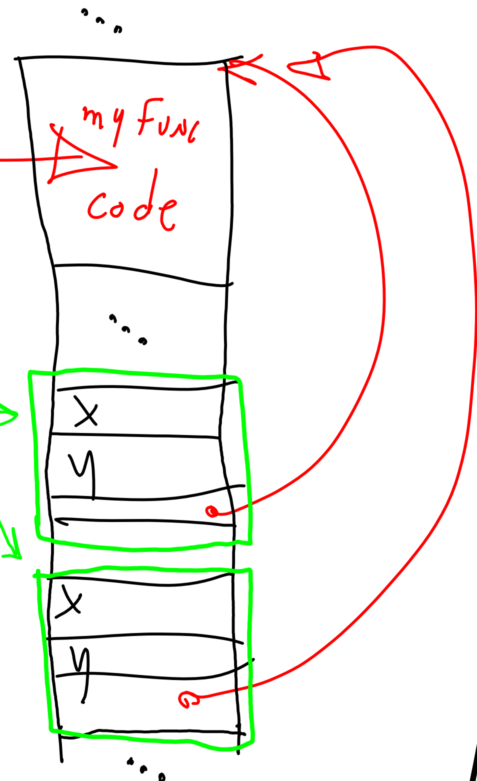
Objects without classes
function pointers

```
struct obj {
    int x
    int (*f)( );
}

struct obj foo;

foo.f = &myFunc;

y = (*(foo.f))( );
```

123456 ⇒ PC fetches instruction 56789

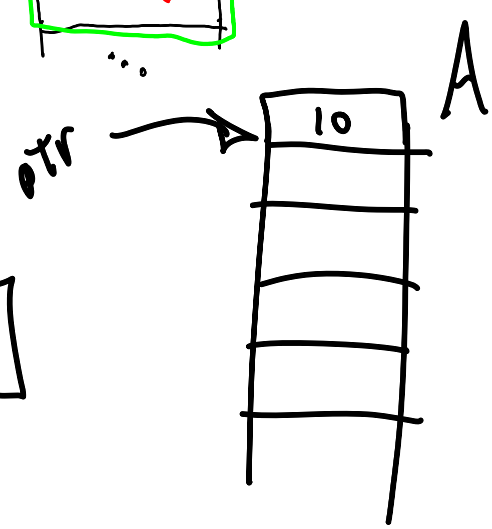



```
int myfunc() {
    code
}
```

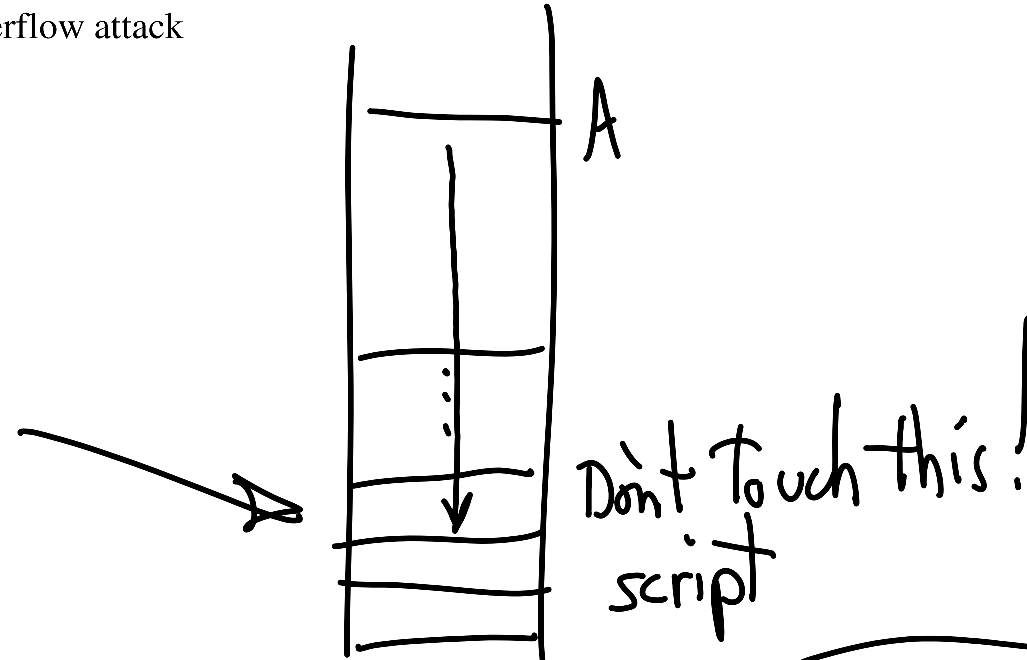


two instances of object, sharing member function via pointers

```
sort(int *list, int (*f)(int,int))
sort(list_ptr, &myFunc)
int A[10];

ptr = &A[0]
ptr = A;

(*ptr)[15]      i
A[15]
```



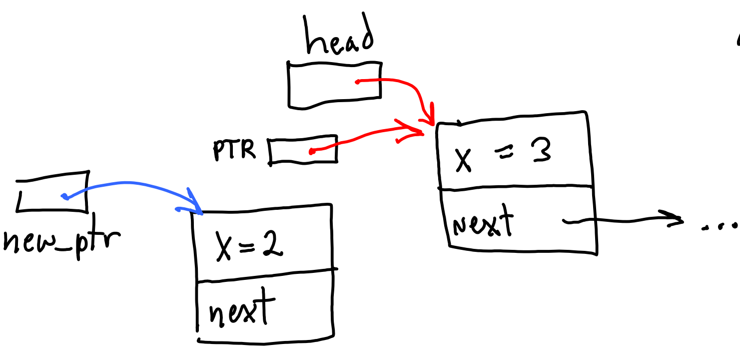
buffer overflow attack



Linked List basics

insert at head

insert before X=3
arg = 3

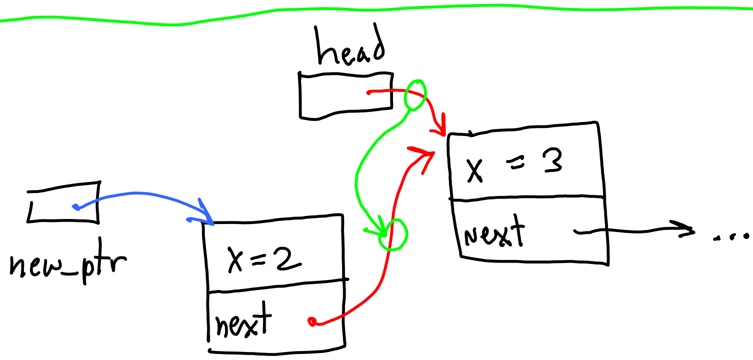


ptr = head

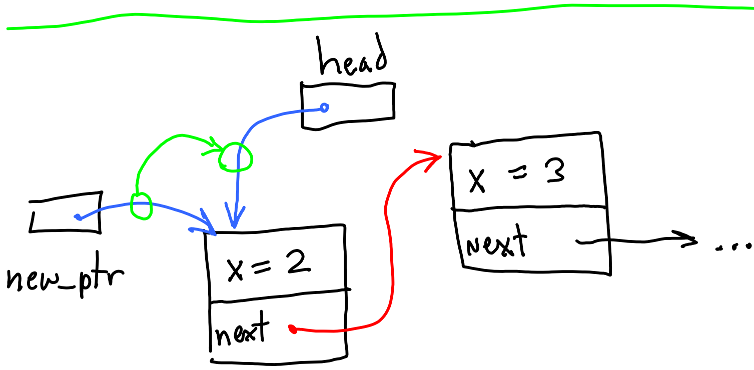
while (ptr != NULL)

if (ptr == head) * (NULL?)

if (ptr -> X == arg)



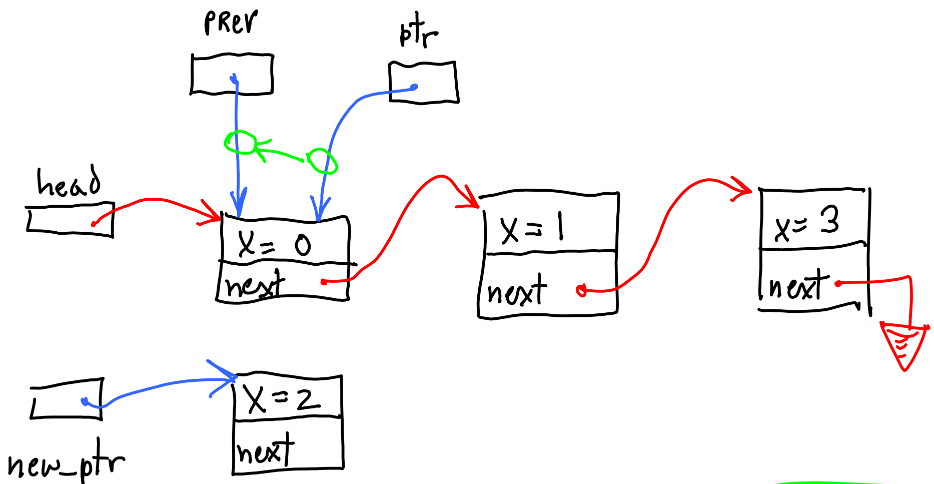
new_ptr -> next = head



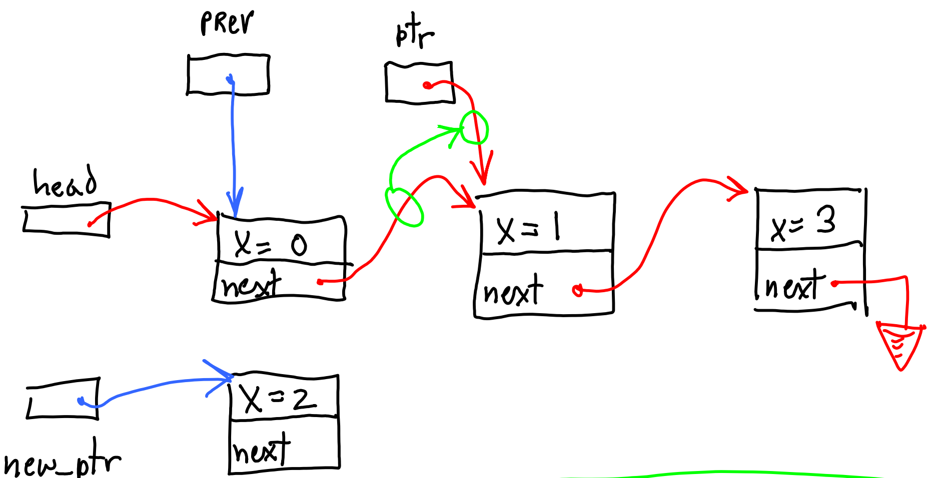
head = new_ptr

break

insert at end



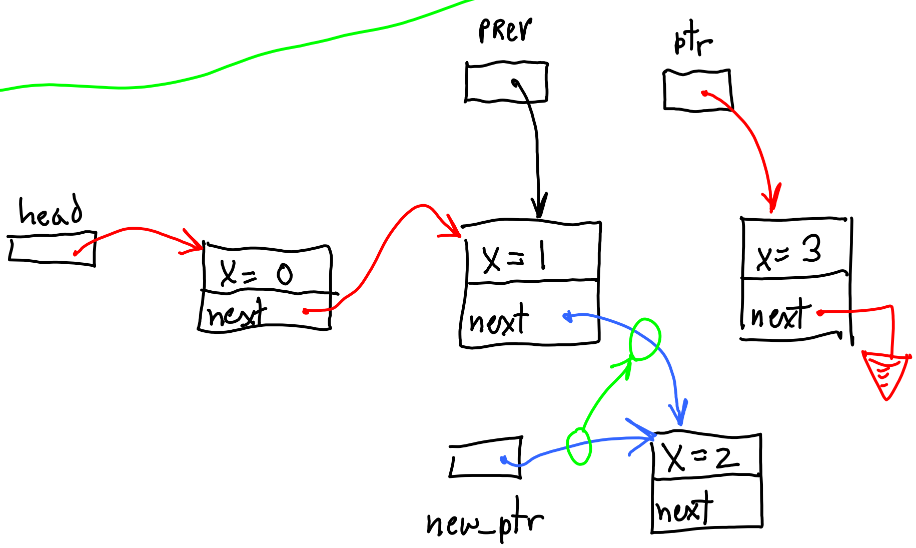

```
ptr = head
while(ptr != null)
    if(ptr == head)
        ...
        break

prev = ptr
```



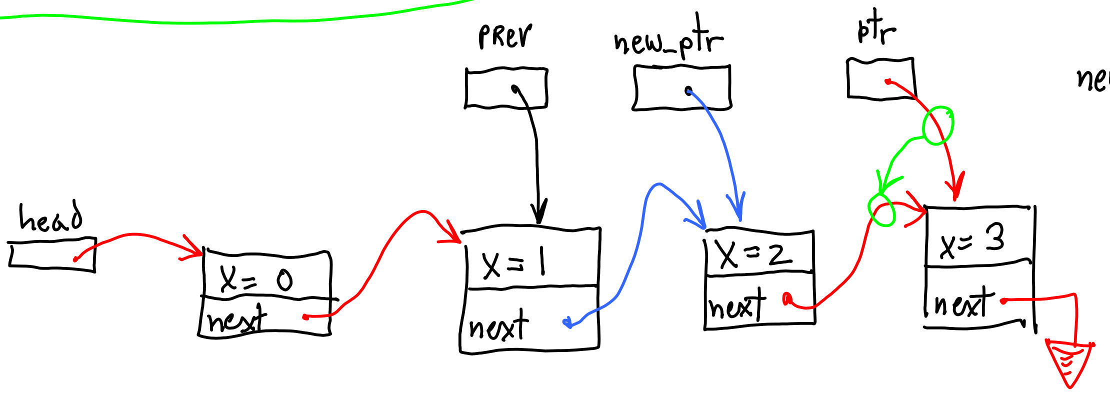

```
ptr = prev->next

if (ptr == NULL)
    ...
    break
```

(come back To this)

```
if (ptr->x == arg)
```

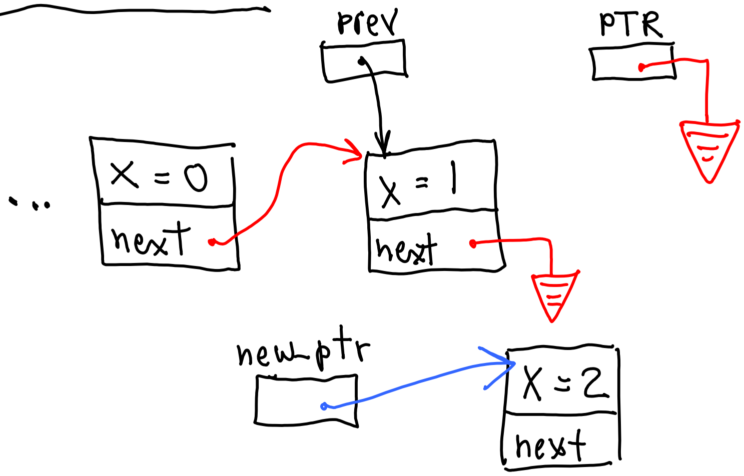



```
prev = ptr
ptr = ptr->next

prev->next = new_ptr
```



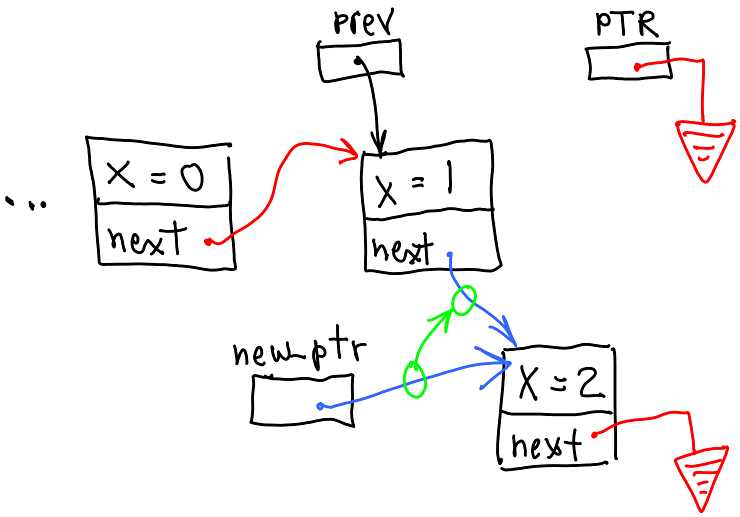

```
new_ptr->next = ptr

break
```

insert at end



if (ptr == NULL)

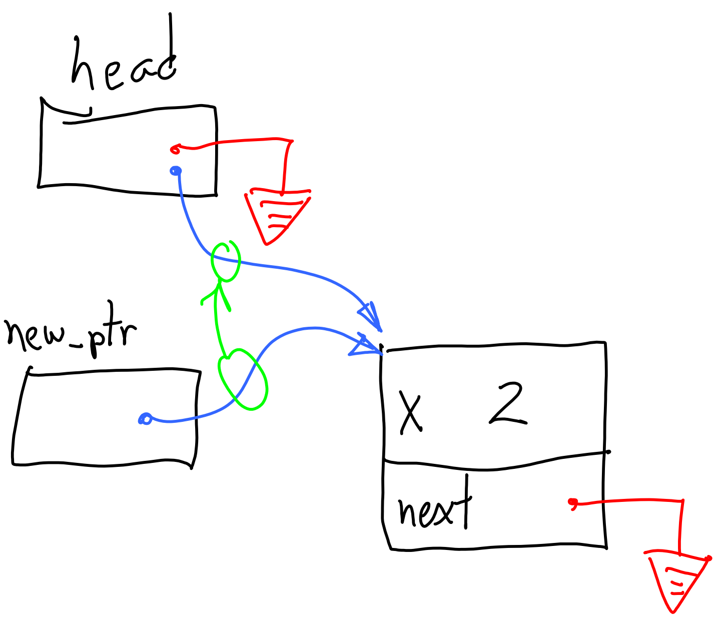
//-- arg not in list



prev->next = new_ptr

new_ptr->next = NULL

* head = NULL yet another special case



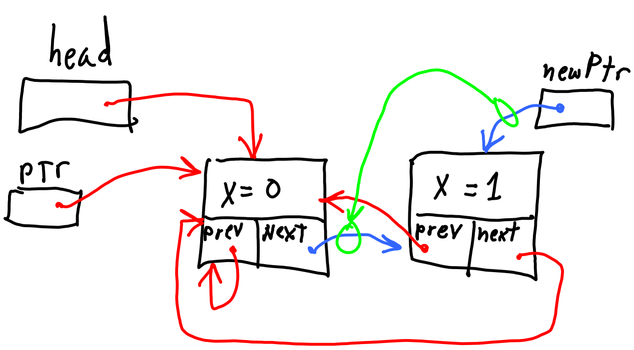
if (head == NULL)

head = new_ptr

head->next = NULL

Doubly-Linked Lists

```
#include "Node.h"
int main(){
    Node * ptr;
    Node * newPtr;
    Node * head;

    //------ make a list
    head = new Node( 0 );   //--- dummy node
    head->next = head->prev = head;

    //--- Insert at head:
    //---    1. find insertion point
    ptr = head;

    //---    2. insert
    newPtr = new Node(1);
    newPtr-> next = ptr->next;
    newPtr->prev = ptr;
    newPtr->prev->next = newPtr;
    newPtr->next->prev = newPtr;

    //--- Insert at end
    //---   1. find insertion point
    ptr = head->prev;

    //---    2. insert
    newPtr = new Node(2);
    newPtr->next = ptr->next;
    newPtr->prev = ptr;
    newPtr->prev->next = newPtr;
    newPtr->next->prev = newPtr;

}
```



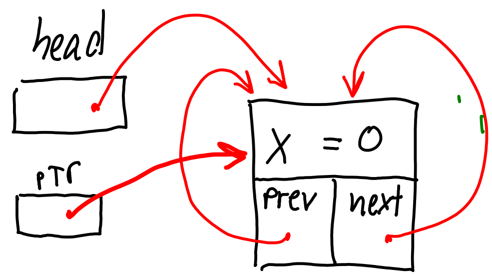

```
class Node {
  public:
      Node( int ); //--constructor
      int x;
      Node *prev;
      Node *next;
};
Node::Node( int val ): {
    x = val;
}
```

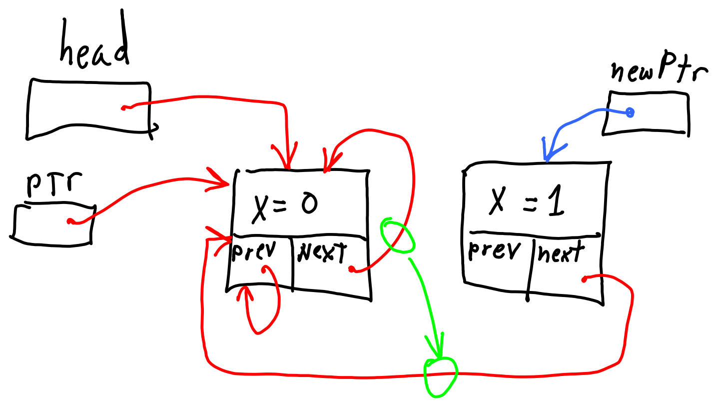
ptr = ptr->next

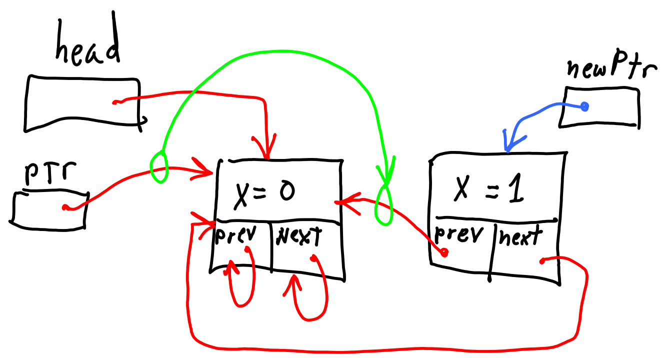
if(ptr == head)

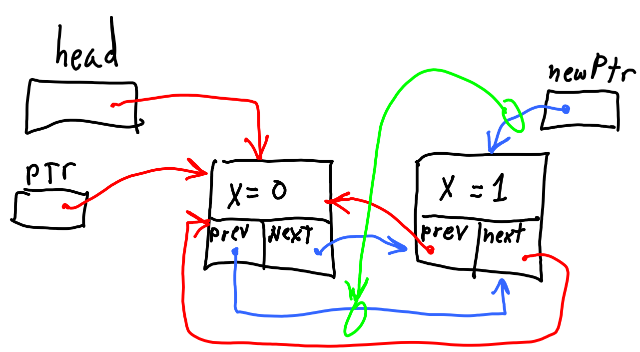
end-of-list

if(== arg)

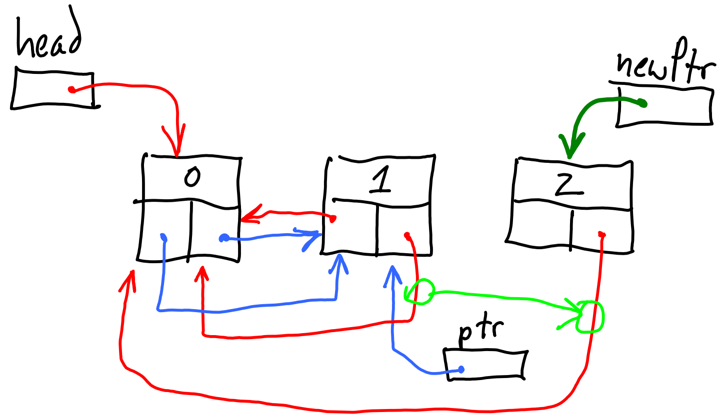


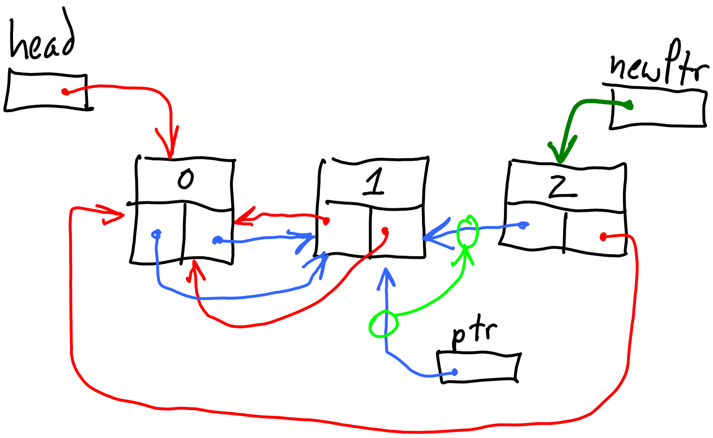


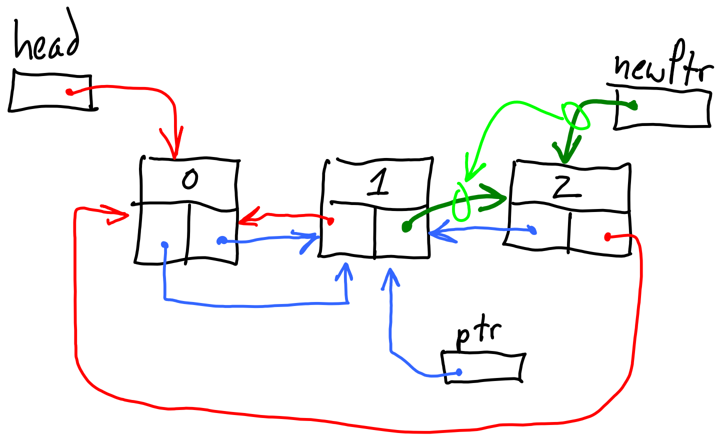




```
newPtr = new Node(2);
   newPtr->next = ptr->next;
   newPtr->prev = ptr;
   newPtr->prev->next = newPtr;
   newPtr->next->prev = newPtr;
```





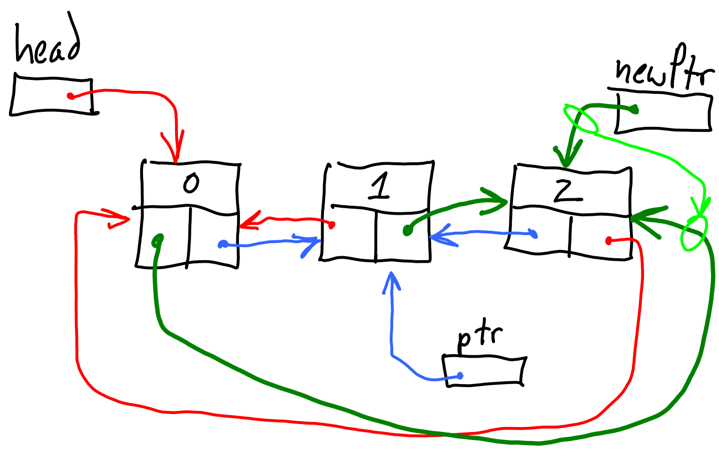


Works for any insertion point.

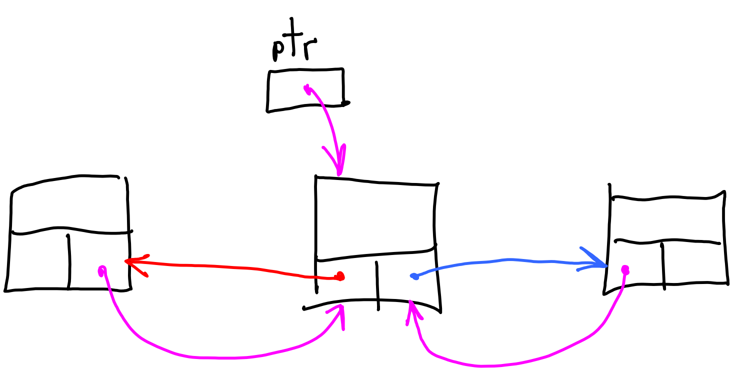
Don't delete head.

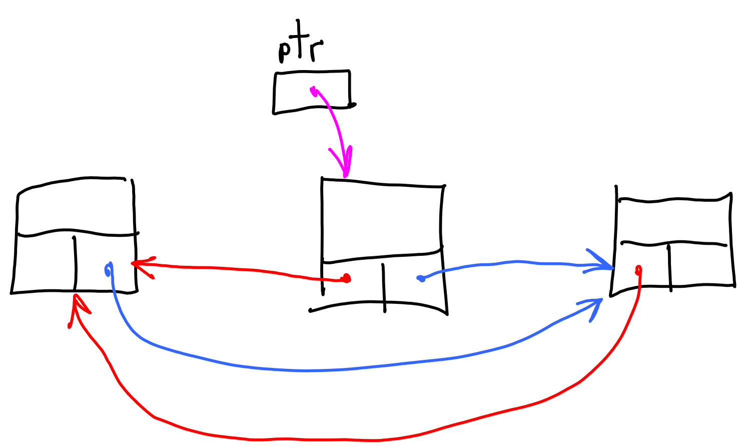
Don't lose pointers (overwrite w/o copy).

Do backwards for deletion:




```
ptr->prev-next = ptr-next
ptr->next->prev = ptr-prev
```





Semantics

class

List {

public

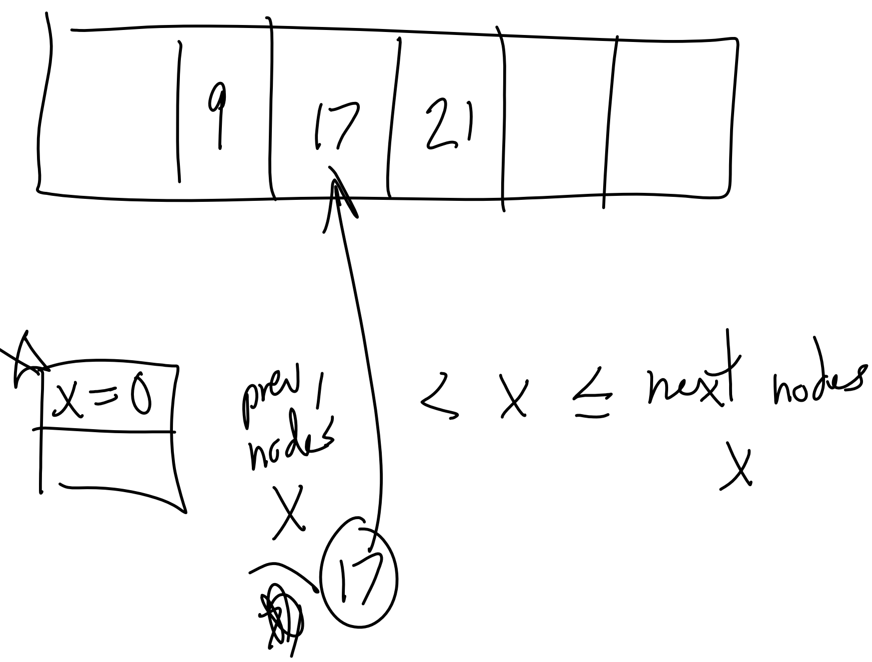
insert(ptr)

delet(ptr)

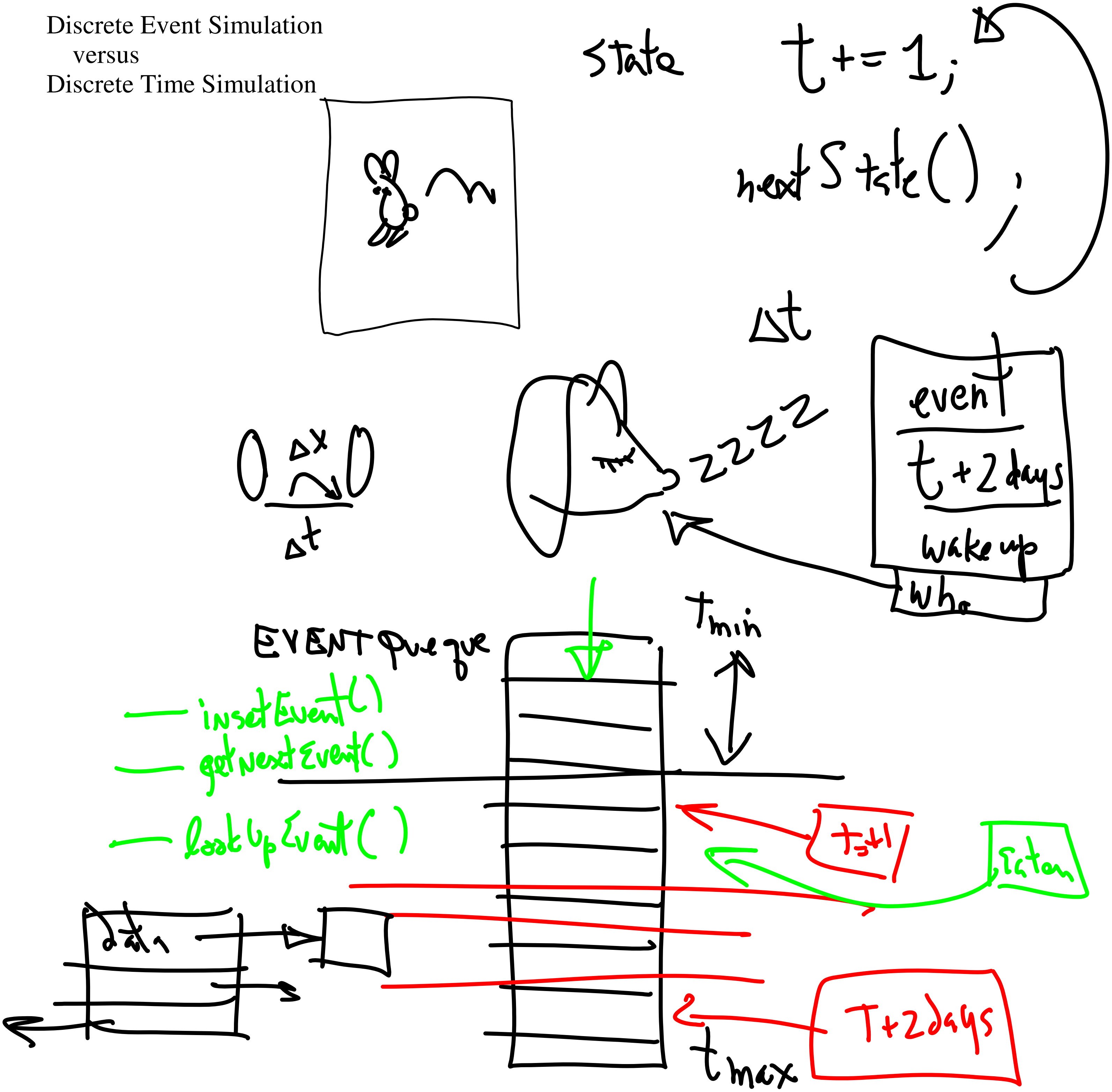
ptr = get( )

}

ordered List



Discrete Event Simulation
versus
Discrete Time Simulation



Tiny PTC

1. fix Makefile
2. get nasm
3. get Xv library (?)

get

numitems = k

while (1) {

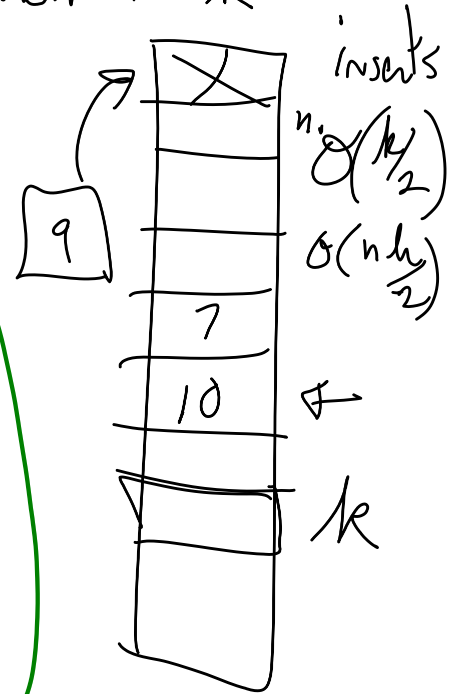
eventPtr = queue.getEvent( )

processEvent(eventPtr);

event = new Event();

queue.insertEvent(event);

} n inserts $n \cdot O(1) = O(n)$ $run(n) \leq k(1)$



$A[numitem+1] = event$

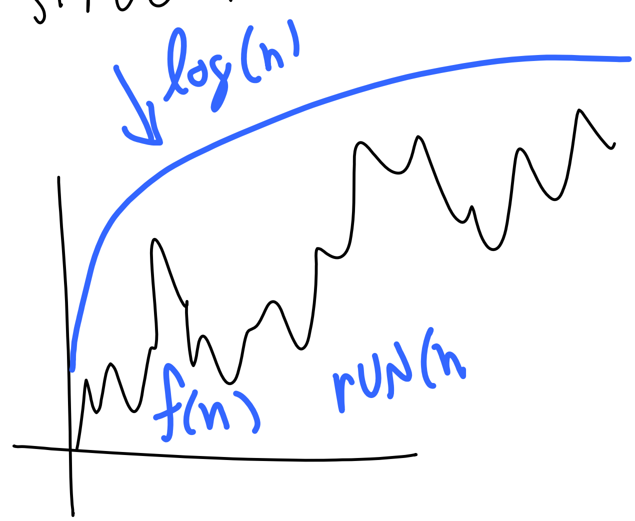
What's a "good" data structure?

gets $n, = m$ $O(k \cdot n)$

$f(n) = O(\ ?\ )$

$g(n)$

$run(n)$

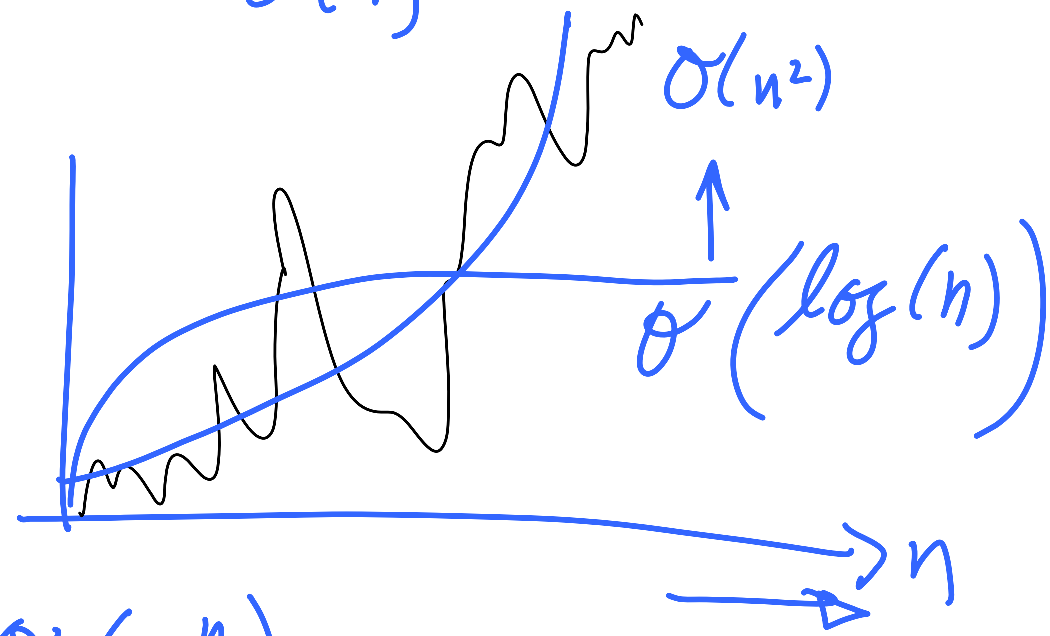


$run(n) = O(\log(n))$

$run(n) \leq c\, g(n) + k_2$

$Run(n) \leq c'n$ $c' = (c \cdot k)$ $\left(\frac{kn}{2}\right)$

$= O(n)$



$run(n) = O(2^n)$

insert → $O(\log(n))$ per operation

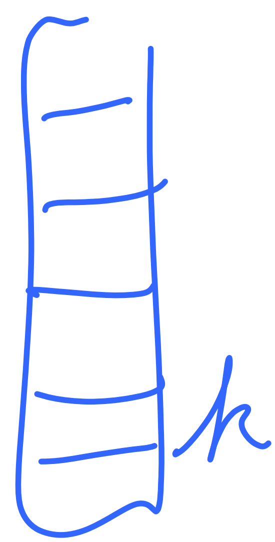
split → $O(\log(n))$

$O(n \cdot \log(n))$

$O(k \cdot n)$

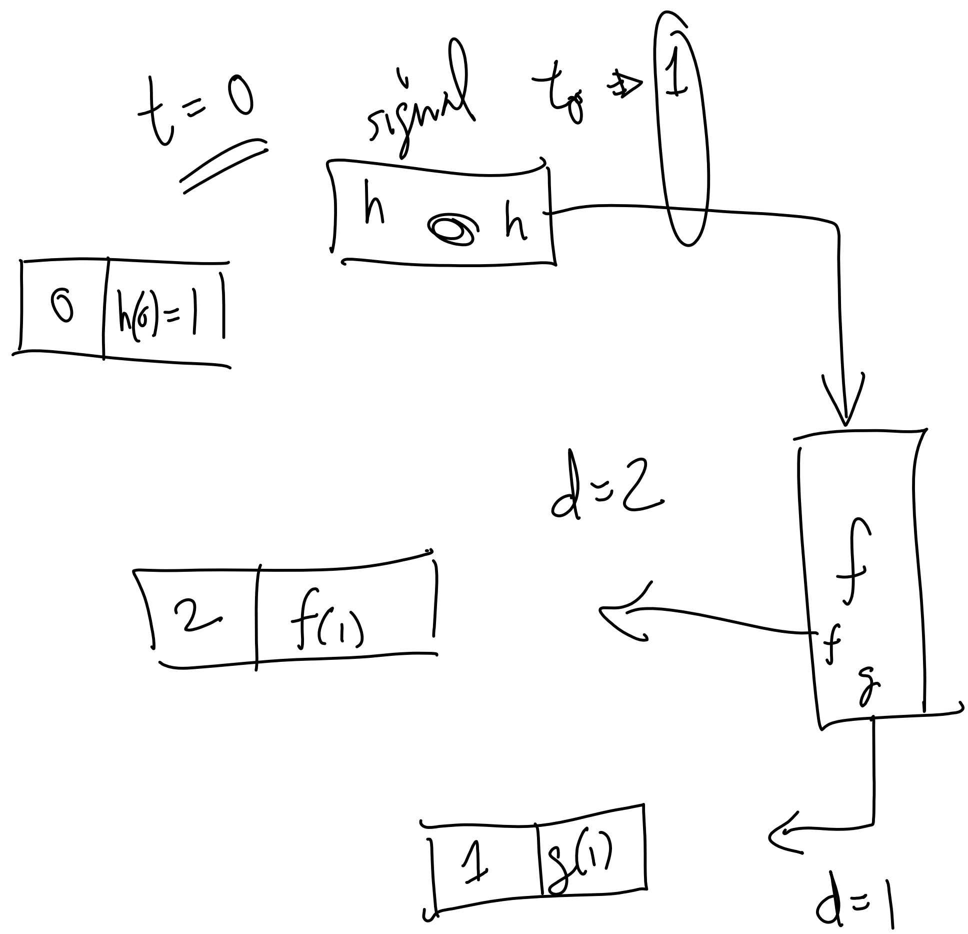
$O(\frac{n}{a} \cdot n)$

$O(n^2)$



$k = \frac{n}{a}$

priority queue



Conflicts

svn up "C"

ci → can't conflict

Take changed files, move to Desktop/tmp

rm -rf mybranch

svn co, svn up

move Desktop/tmp → branch

svn status
svn up

svn ci