Lecture 3

Arrays and Linked Lists

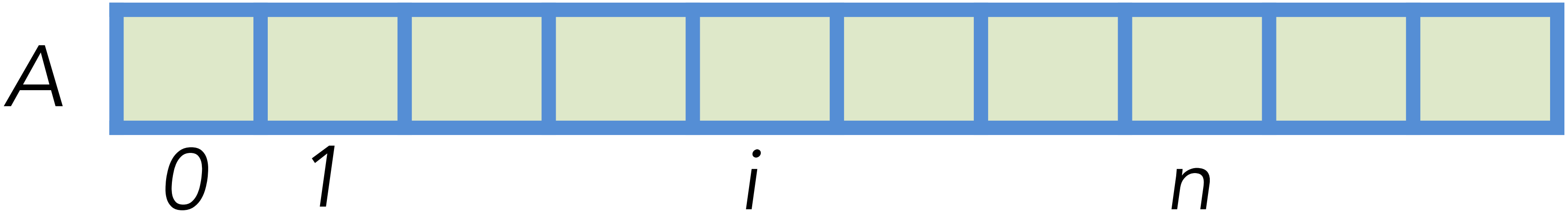
Student Record

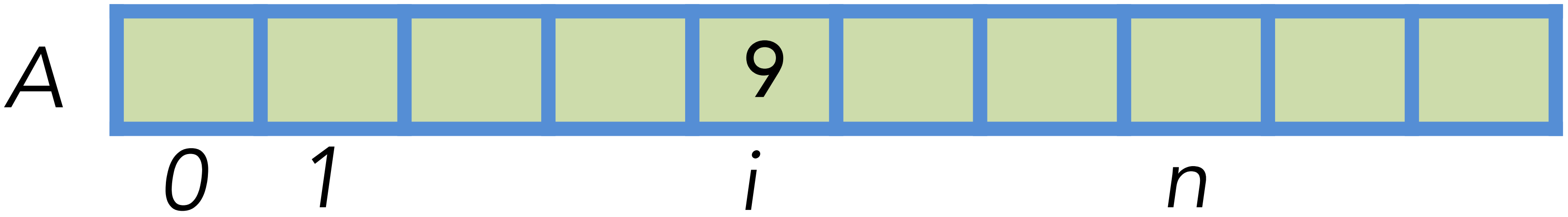
```
public class Student {
    private:
        int ID;
        double score;
        …
    public:
        double getScore(){
            return score;
        }
        …
}
```

```
Student Mohan;
Student Sohan;
```

Arrays

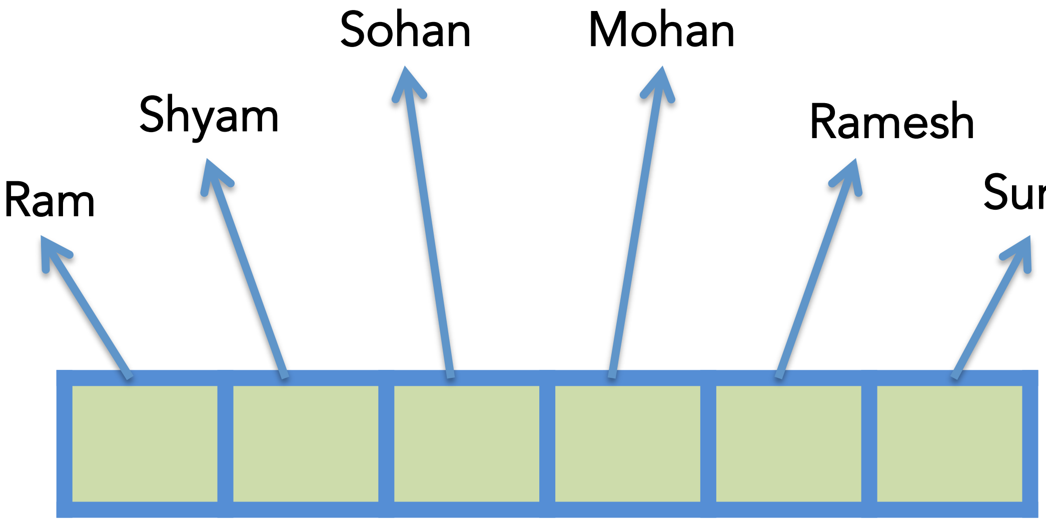
sequenced collection of variables of the same type





```
int[] A = {8, 2, 5...};
A[i] = 9;
```

```
Student* st_list[100];
for  (int i=0;i++;i<100)
    st_list[i]=new Student(ID, score);
double score = st_list[i]->getScore();
```



Array of Objects

8	2	5	7	3	8

Array of Integers

What operations are performed on an array?

Sorting

Min/Max

Addition/Deletion

Sorting

Insertion Sort

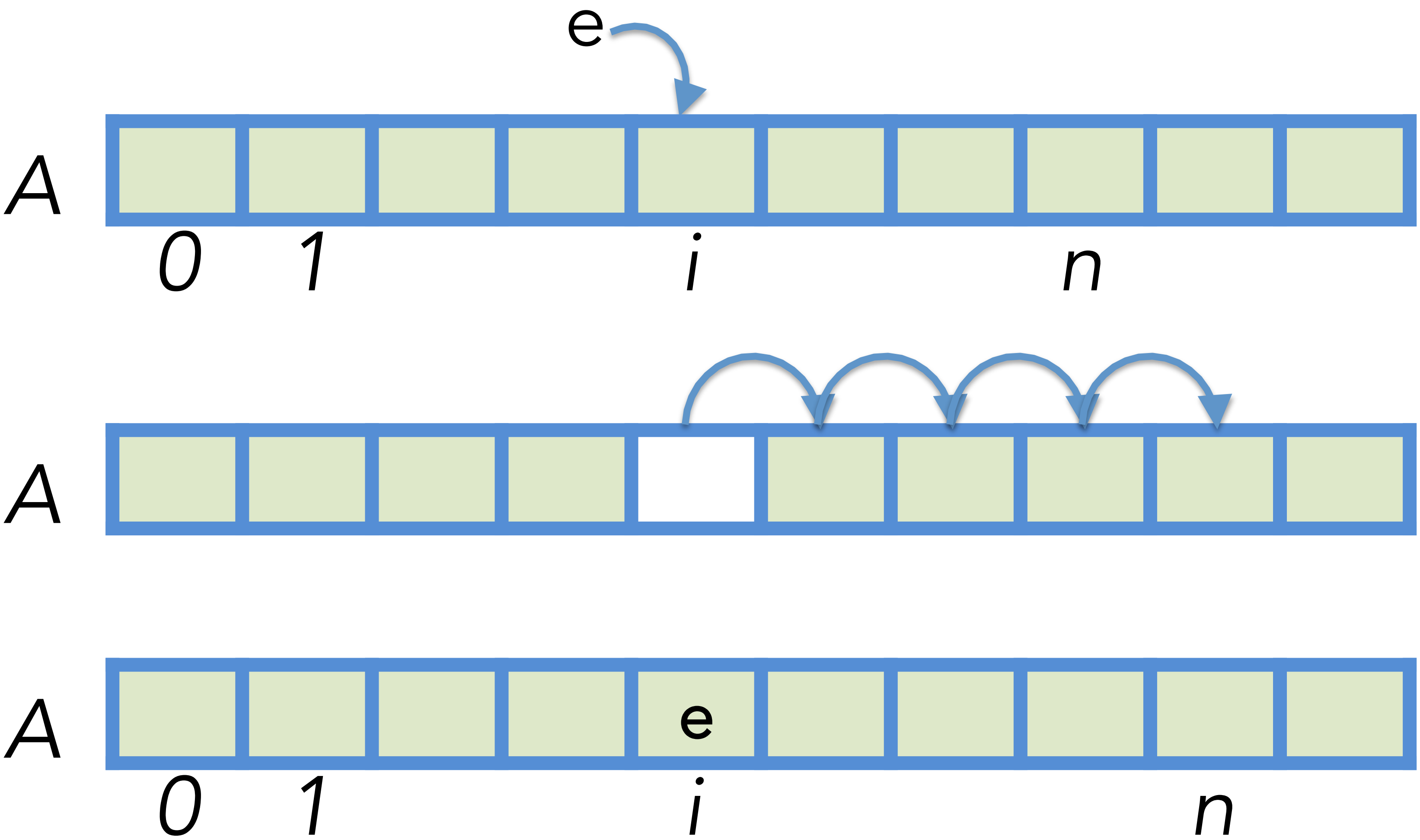
Insert elements at right place one by one!

5	2	7	8	4	1	9

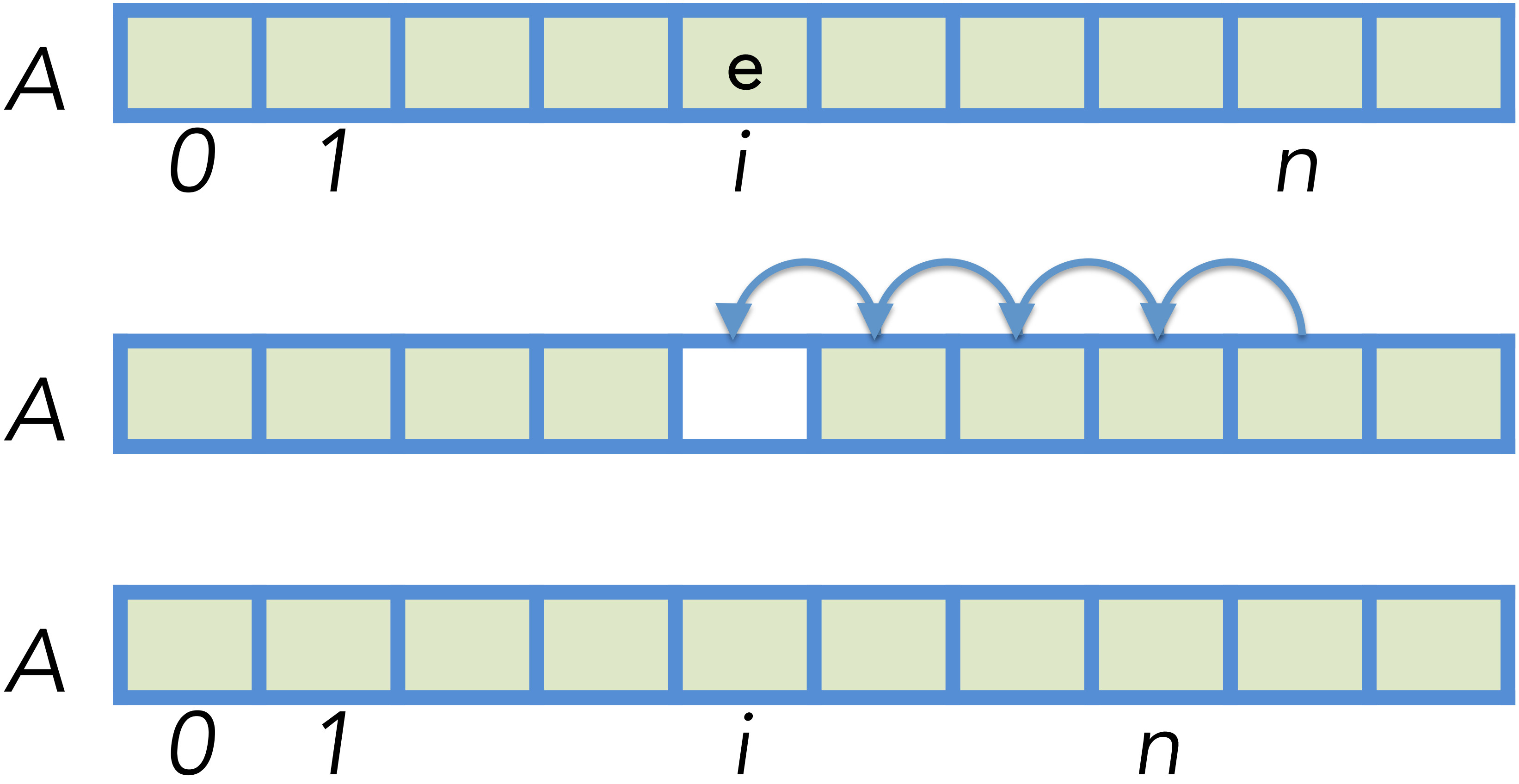
Min/Max

Unsorted	5	2	7	8	4	1	9
Sorted	1	2	4	5	7	8	9

Addition



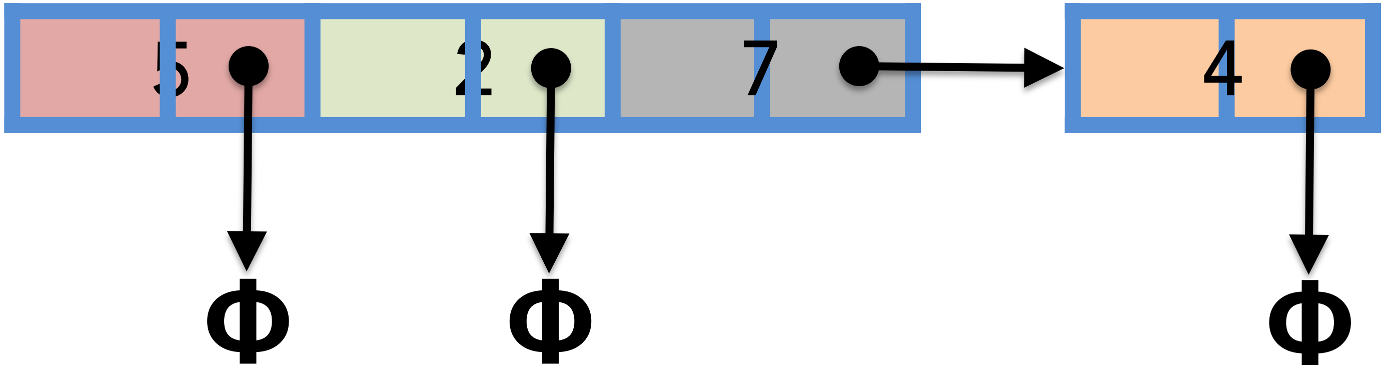
Deletion



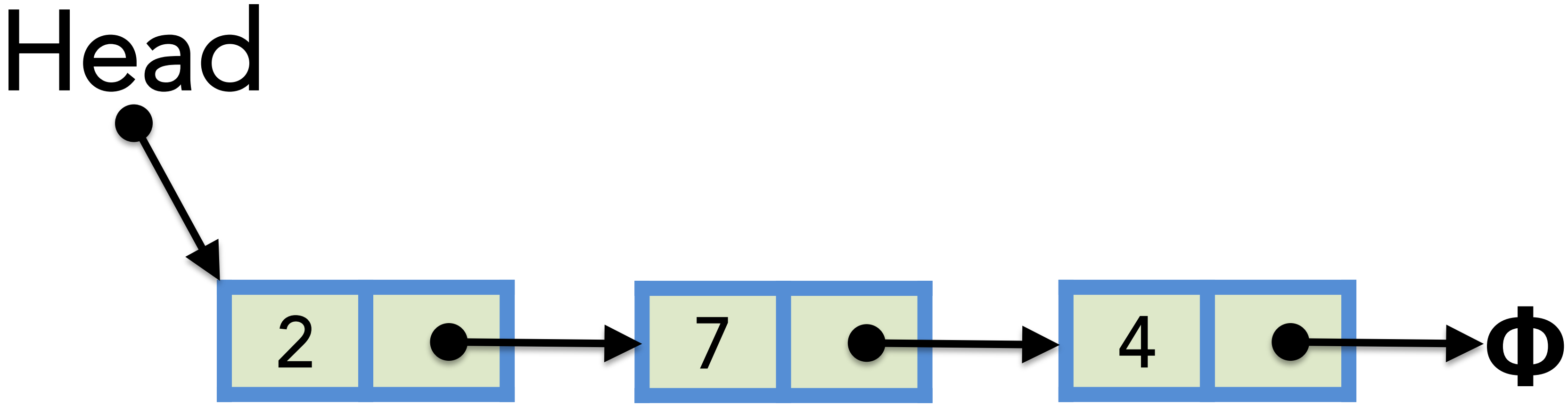
Array Limitations?

1. Fixed capacity
2. Empty cells
3. Expensive Addition/ Removal

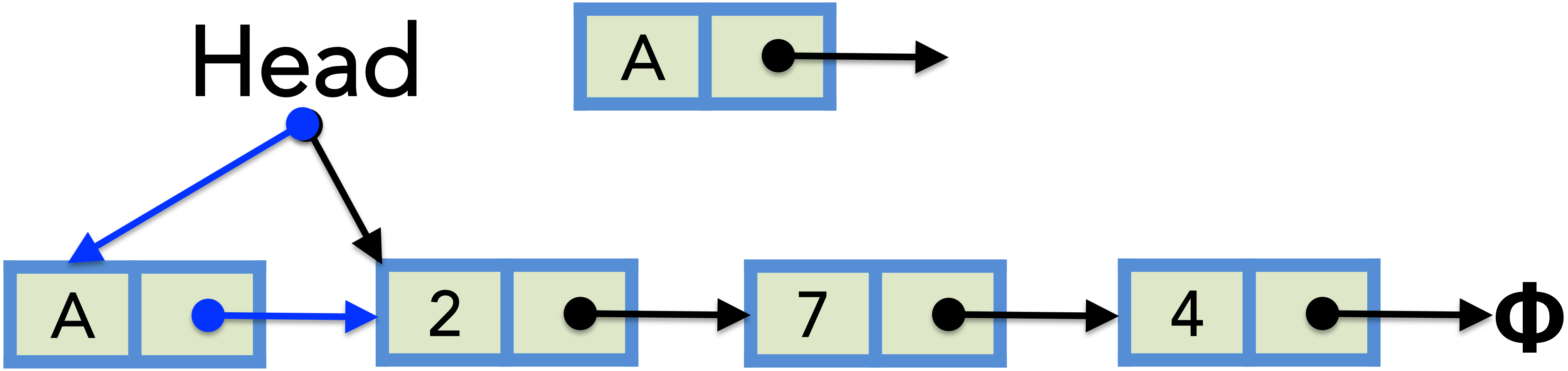
How to add new element?



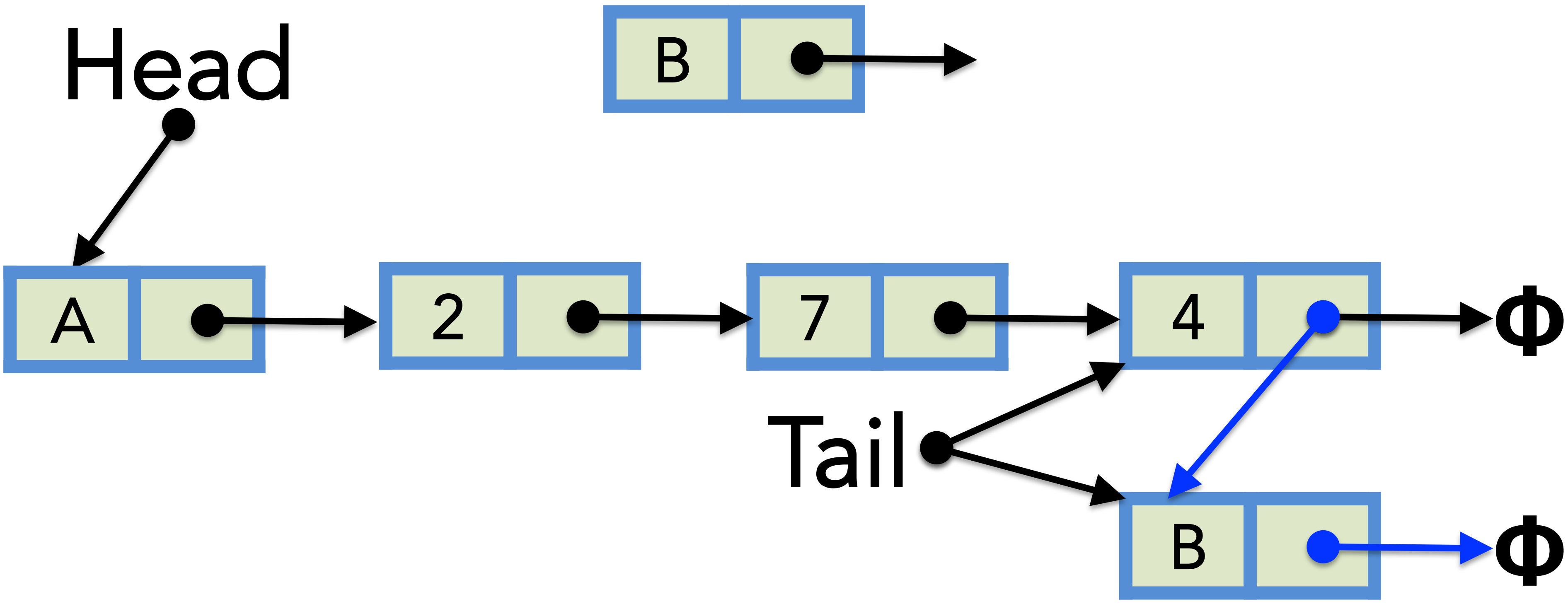
Linked List



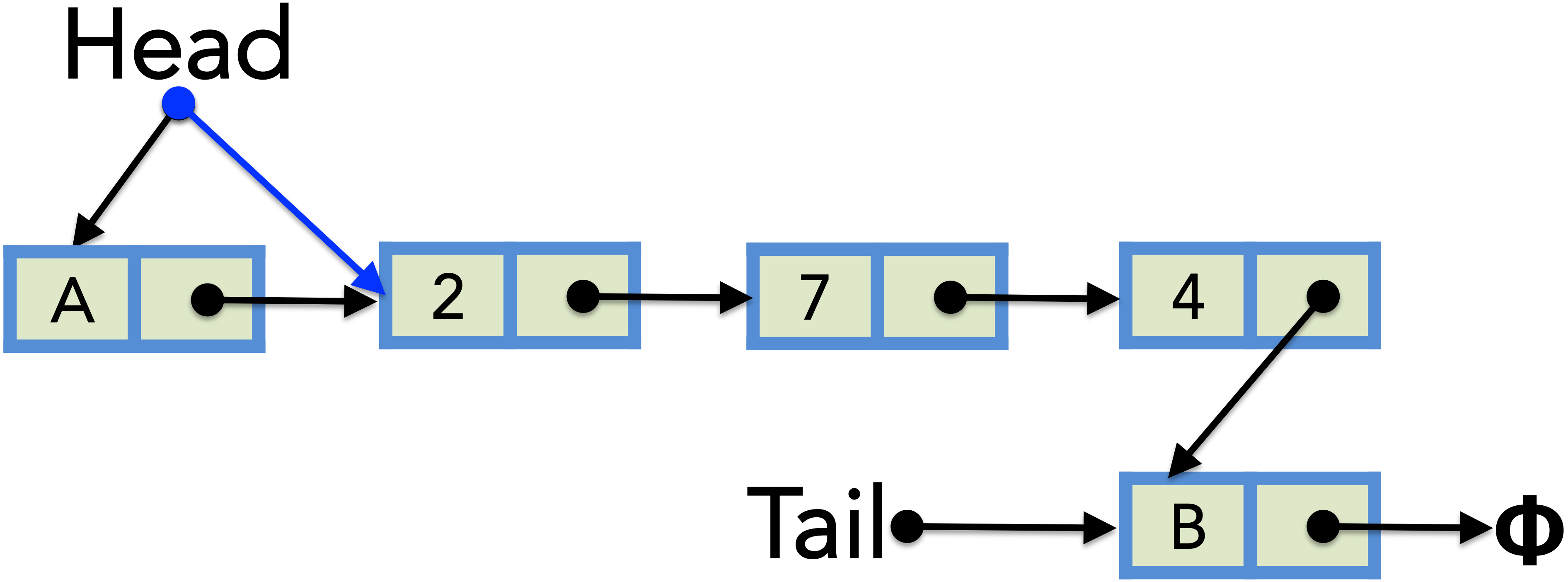
Add A at Head



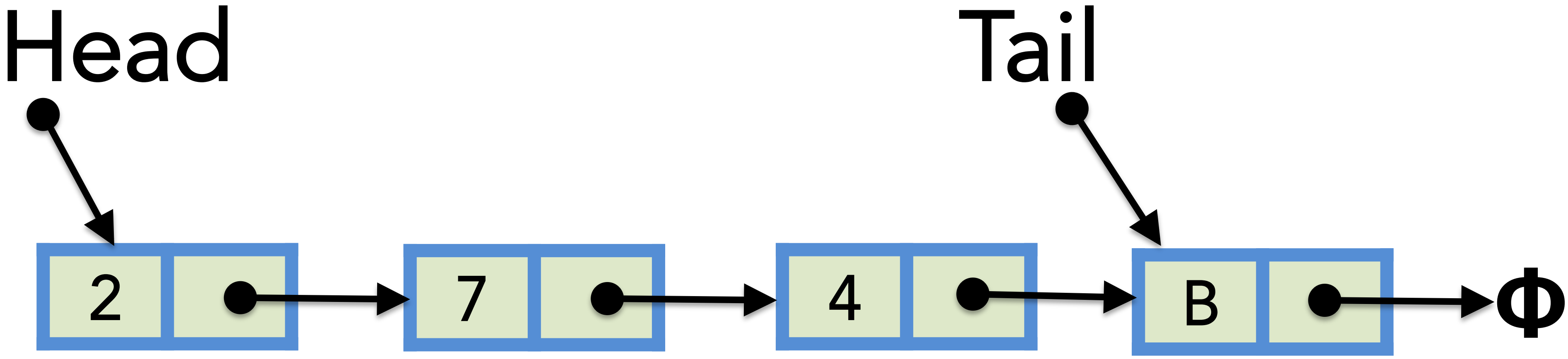
Add B at the End



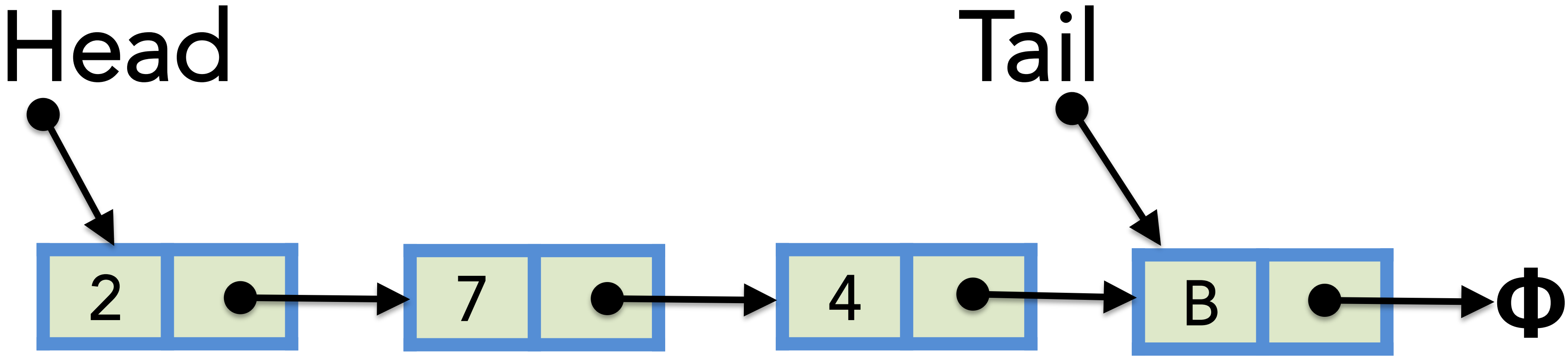
Remove A



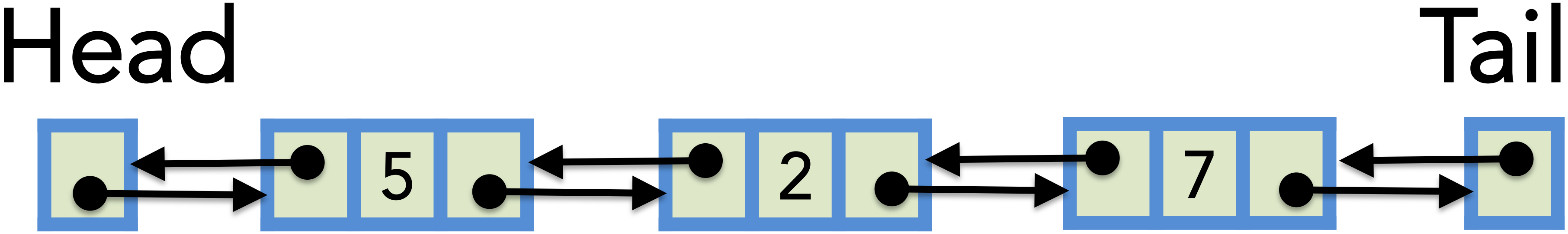
Remove B



Remove B

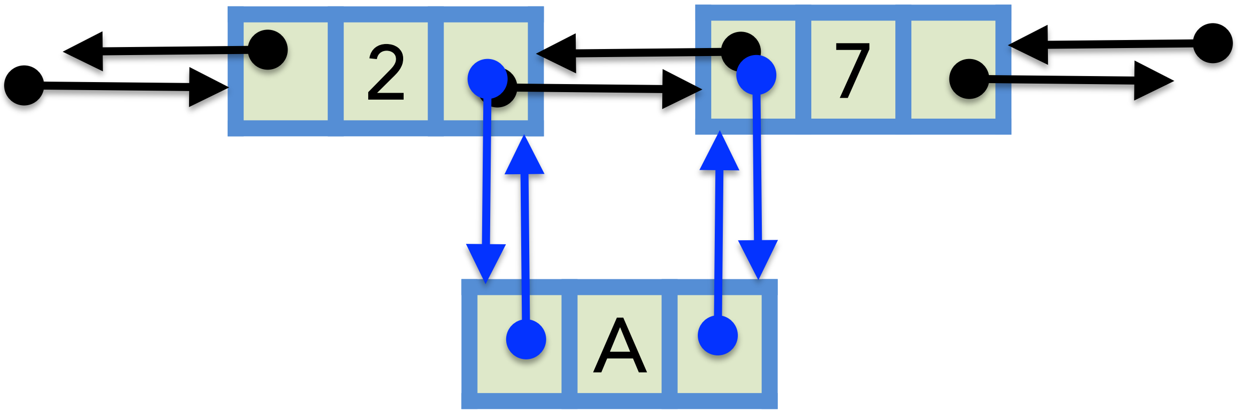


Doubly Linked List

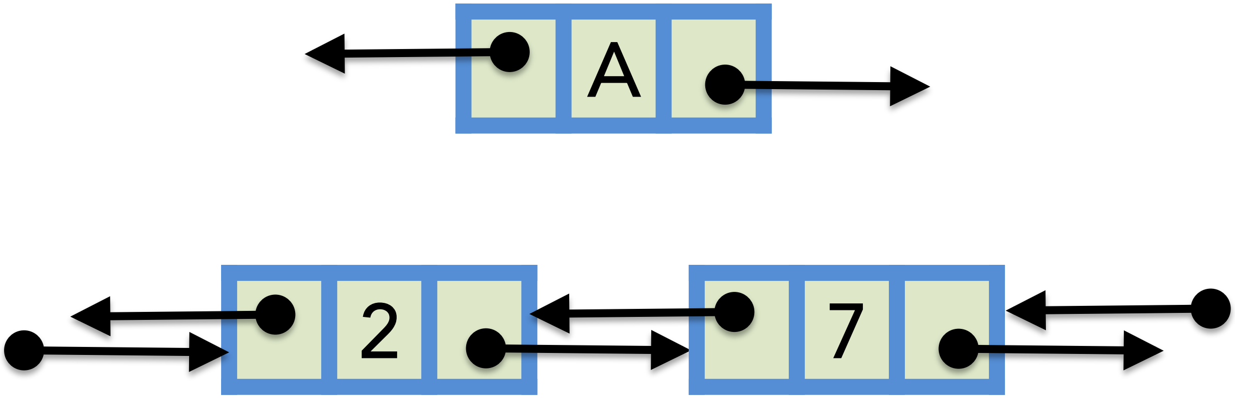


{next, E, prev}

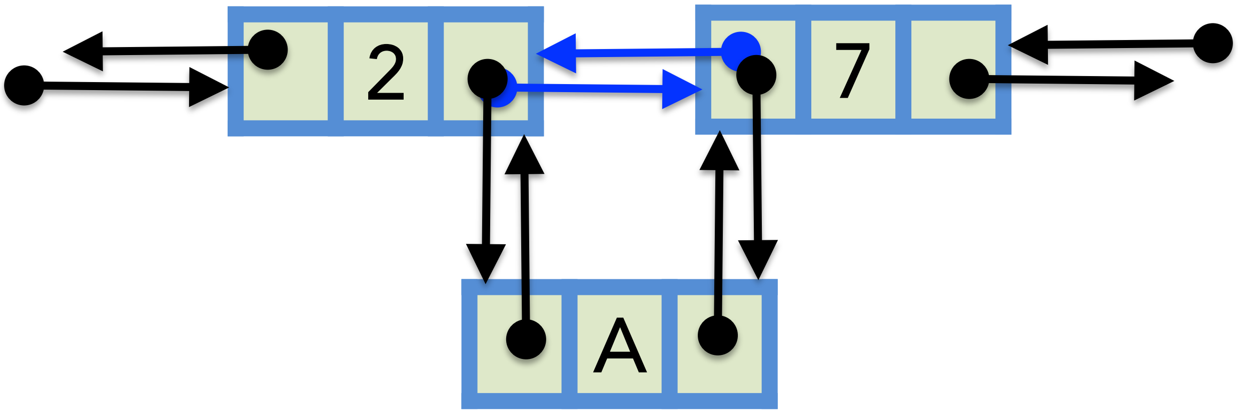
Insert A



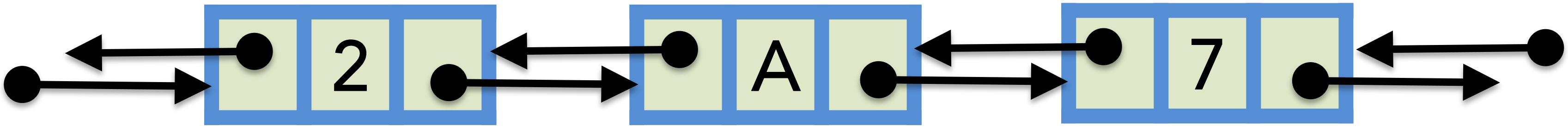
Insert A



Remove A



Remove A



Where do we use linked list?

- constant time addition/deletion
- number of items not known
- don’t need random access
- insert anywhere

Disadvantages with respect to arrays?

A Linked List Node

```
class StringNode {
private:
  string elem;
  StringNode* next;

  friend class StringLinkedList;
};
```

Linked List Class

```
class StringLinkedList {
public:
  StringLinkedList();
  ˜StringLinkedList();
  bool empty() const;
  const string& front() const;
  void addFront(const string& e);
  void removeFront();
private:
  StringNode* head;
};
```

Bookkeeping

```
StringLinkedList::StringLinkedList()
  : head(NULL) { }

StringLinkedList::˜StringLinkedList()
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const
  { return head == NULL; }

const string& StringLinkedList::front() const
  { return head–>elem; }
```

Add at Front

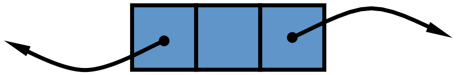
```
void StringLinkedList::addFront(const string& e) {
  StringNode* v = new StringNode;
  v->elem = e;
  v->next = head;
  head = v;
}
```

Remove from Front

```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old–>next;
  delete old;
}
```

Doubly Linked List Node

```
class DNode{
private:
    Elem elem;
    DNode* prev;
    DNode* next;
    friend class DLinkedList;
}
```

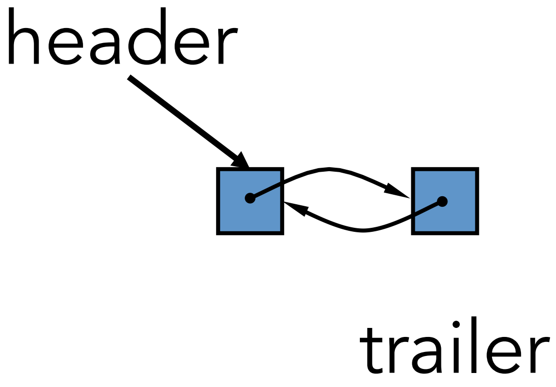


Doubly Linked List Class

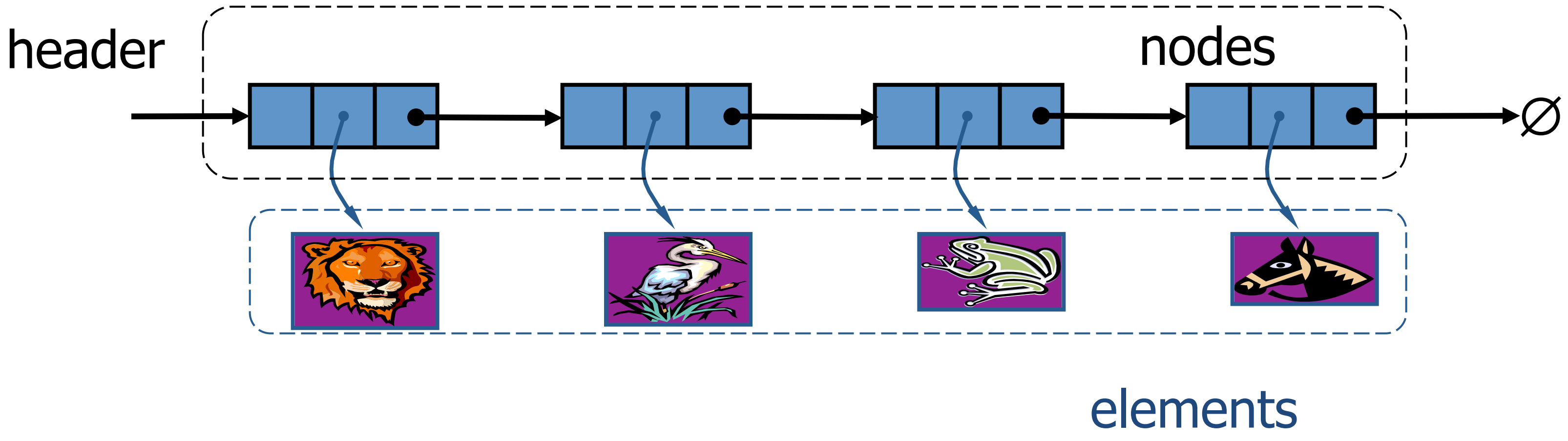
```
class DLinkedList{
    public:
            DLinkedList();
           ~DLinkedList();
            bool empty() const;
            const Elem& front() const;
            const Elem& back() const;
            void addFront(const Elem& e);
            void addBack(const Elem& e);
            void removeFront();
            void removeBack();
    private:
            DNode* header;
            DNode* trailer;
    protected:
            void add(DNode* v, const Elem& e);
             void remove(DNode* v);
};
```

Constructor

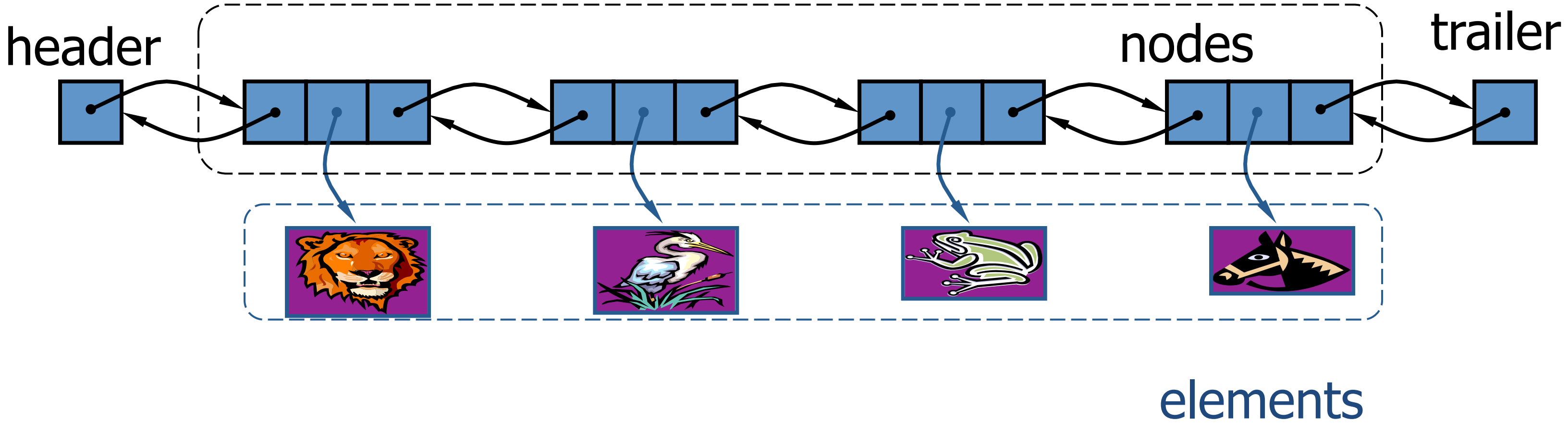
```
DLinkedList::DLinkedList(){
    header = new DNode;
    trailer = new DNode;
    header->next = trailer;
    trailer-> prev = header;
}
```



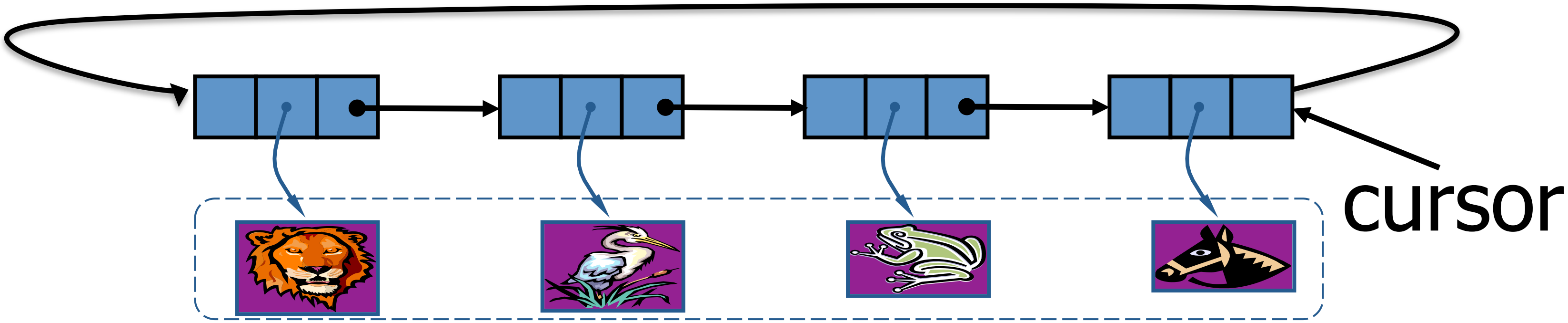
Singly linked list



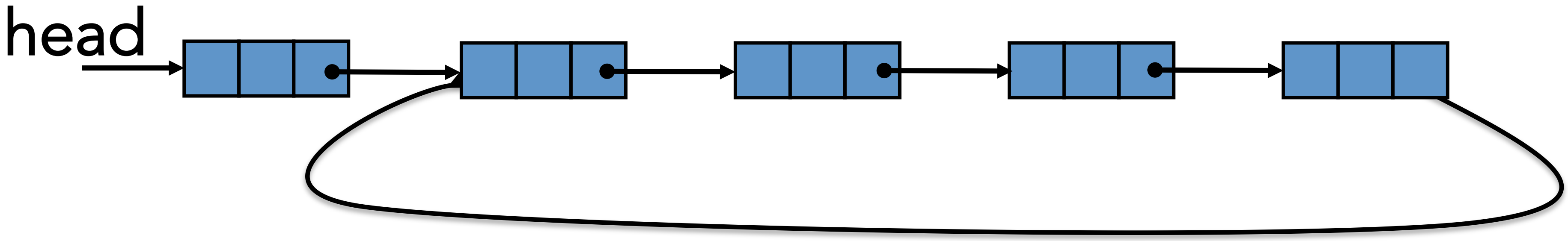
Doubly linked list



Circular Linked Lists



Given the head, how will you find that there is a cycle in the list?



Solutions

- Traverse until end?
- Traverse until find head again?
- Mark each node?
- Create list of nodes visited so far and match the current node!
- Reverse the list
- Fas-slow iterators