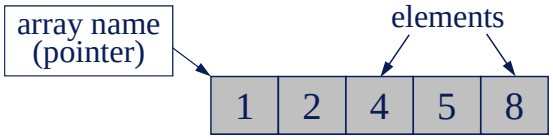
Linked List vs Array

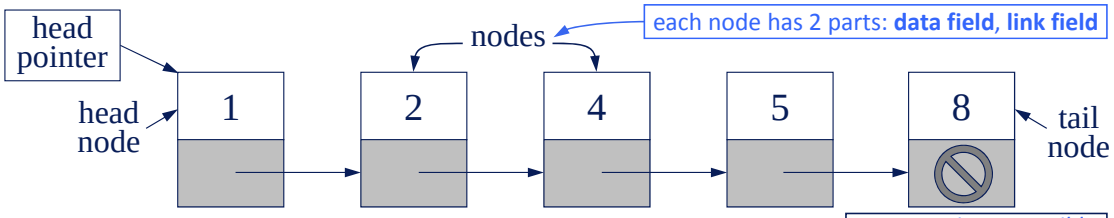
- Similarity:
  - Container data structure for storing collection of data items



- Differences:
  - Array
    - Consists of a *single block* of *contiguous* memory locations
    - ***Elements*** form a logical group by being *next to each other*
  - Linked list
    - Consists of as *many blocks* of *separate* memory locations as there are data items
    - ***Nodes*** form a logical group by being *linked in some way through their addresses*

allocated at compile time or at run time

usually allocated at run time



☞ For linked list shown above → called ***singly-linked list***: *many variants possible*

- Each node holds a *data item* (in data field) and a *pointer to next node* (in link field)
- Special "end marker" value for last node's link field

Simple node structure (good for instructional purpose):

```
struct Node
{
   int data;
   Node* link;
};
```

Use of *null address* as special situation signal values (good for instructional purpose):

- Address in *head pointer of empty list*.
- Address in *link field of tail node*.

Due to the linking structure, linked lists behave differently than arrays; tabulated below are the key differences:

	Array	Linked List
get/set positional access	getting/setting an element at some specified position is $O(1)$	getting/setting a node at some specified position is $O(n)$ because we must go through a sequence of links to get to the indicated position
add/remove at first and last positions	adding at **last** position is $O(1)$ in general; an additional $O(n)$ cost is incurred when array has to be resized; if resizing is managed efficiently, overall average cost is still $O(1)$ --- removing at **last** position is always $O(1)$ --- adding/removing at **first** position is always $O(n)$ because rest of array has to be shifted	adding at first position is $O(1)$; removing at **last** position is also $O(1)$ with doubly-linked lists
add/remove at arbitrary position	adding/removing at arbitrary position is $O(n)$ due to shifting	adding/removing at arbitrary position is $O(n)$ due to sequencing through links
space usage	wasted space varies according to unused array elements; in order to keep average add time efficient, $O(n)$ wasted space typically incurred when resizing	links are "wasted" by virtue of not holding data; thus there is always $O(n)$ wasted space

In rather general terms:

- `array` is better choice when we need random positional access of data items whereas `linked list` is better choice when we need sequential positional access (including end-based access) of data items.
- `array` tends to be advantageous when there's little need for adding/removing data items (little "data come data go") whereas `linked list` tends to be advantageous when there's much need for adding/removing data items (much "data come data go").

Examples Illustrating Common Idioms/Pitfalls

```
void ShowList(Node* head)
{
   Node* cur = head;
   while (cur != 0)
   {
      cout << cur->data << endl;
      cur = cur->link;
   }
}
```

```
// is this OK?
void ShowList(Node* head)
{
   while (head != 0)
   {
      cout << head->data << endl;
      head = head->link;
   }
}
```

```
// is this OK?
void ShowList(Node*& head)
{
   while (head != 0)
   {
      cout << head->data << endl;
      head = head->link;
   }
}
```

Why not simply adapt logic of `ShowList`'s body?

```
void DestroyList(Node*& head)
{
   Node* nodePtr = 0;
   while (head != 0)
   {
      nodePtr = head->link;
      delete head;
      head = nodePtr;
   }
}
```

```
// is this OK?
void DestroyList(Node* head)
{
   Node* nodePtr = 0;
   while (head != 0)
   {
      nodePtr = head->link;
      delete head;
      head = nodePtr;
   }
}
```

```
// is this OK?
void DestroyList(Node*& head)
{
   Node* cur = head;
   Node* nodePtr = 0;
   while (cur != 0)
   {
      nodePtr = cur->link;
      delete cur;
      cur = nodePtr;
   }
}
```

```
void AppendNode(Node*& head, int newVal)
{
   Node* nodePtr = new Node;
   nodePtr->data = newVal;
   nodePtr->link = 0;
   if (head == 0)
   {
      head = nodePtr;
      return;
   }
   Node* cur = head;
   while (cur->link != 0) cur = cur->link;
   cur->link = nodePtr;
}
```

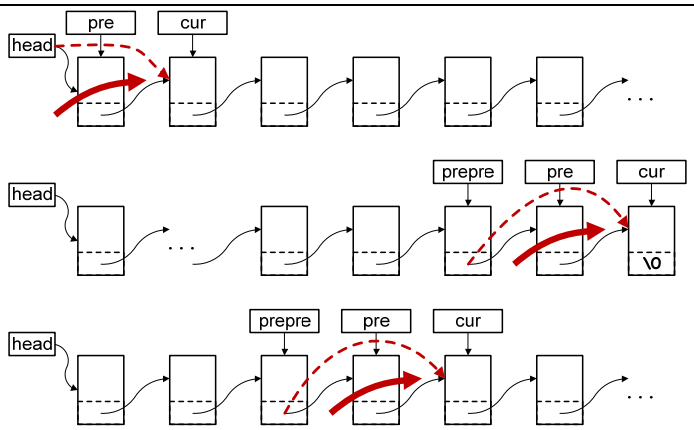
```
// is this OK?
void AppendNode(Node* head, int newVal)
{
   Node* nodePtr = new Node;
   nodePtr->data = newVal;
   nodePtr->link = 0;
   if (head == 0)
   {
      head = nodePtr;
      return;
   }
   Node* cur = head;
   while (cur->link != 0) cur = cur->link;
   cur->link = nodePtr;
}
```

```
bool IsSortedUp(Node* head)
{
   if (head == 0 ||
       head->link == 0) // empty or 1-node
      return true;
   while (head->link != 0) // not @ last node
   {
      if (head->link->data < head->data)
         return false;
      head = head->link;
   }
   return true;
}
```

```
void InsertSortedUp(Node*& head, int value)
{
   Node *pre = 0,*cur = head;
   while (cur != 0 && cur->data < value)
   {  pre = cur;  cur = cur->link; }

   Node *newNodePtr = new Node;
   newNodePtr->data = value;
   newNodePtr->link = cur;
   if (cur == head) head = newNodePtr;
   else pre->link = newNodePtr;
}
```

```
bool DelFirstKeyNode(Node*& head, int key)
{
   Node *pre = 0, *cur = head;
   while (cur != 0 && cur->data != key)
   {  pre = cur;  cur = cur->link;  }
   if (cur == 0)
   {
      cout << key << " not found." << endl;
      return false;
   }
   if (cur == head) //OR pre == 0
      head = head->link;
   else pre->link = cur->link;
   delete cur;
   return true;
}
```




```
bool DelNodeBefore1stMatch(Node*& head, int key)
{
   if (head == 0 || head->link == 0 || head->data == key)
      return false;
   Node *cur = head->link, *pre = head, *prepre = 0;
   while (cur != 0 && cur->data != key)
   { prepre = pre;  pre = cur;  cur = cur->link; }
   if (cur == 0) return false;
   if (cur == head->link) { head = cur;  delete pre; }
   else { prepre->link = cur; delete pre; }
   return true;
}
```

Common Pointer/Memory Woes

- How head pointer(s) should be passed to a function: *by value* or *by reference*.
- Be cognizant of and know how to avoid *null-pointer exception*.
  - » When writing code that dereferences a pointer (at some point), always check that the pointer will never contain the *null address* (at that point).
  - » Be careful when writing relational expression involving *short-circuit evaluation*.
- Know when to say no:
  - » Don't use memory that's not allocated.
  - » Don't access memory already freed up (if need to access it, do so *before* freeing it up).
  - » Don't leak away memory.