```
struct node {
    int data;
    struct node* next;
};
```

H Den postdefinition vi använder i detta exempel

```
int Length(struct node* head) {
    int count = 0;
    struct node* current = head;
    while (current != NULL) {
        count++;
        current=current->next;
    }
    return(count);
}
```

H Returnerar antalet element i listan.

```
int Count(struct node* head, int searchFor) {
    struct node* current=head;
    int count=0;
    while (current!=NULL) {
        if (current->data==searchFor) count++;
        current=current->next;
    }
    return count;
}
```

H Räknar antalet element i listan med värdet `searchFr`

```
int Count2(struct node* head, int searchFor) {
    struct node* current;
    int count = 0;
    for (current=head; current!=NULL; current=current->next) {
        if (current->data==searchFor) count++;
    }
    return count;
}
```

H Så kan man använda en `for`-loop istället för en `while`.

```
1 void Push(struct node** headRef, int newData) {
2     struct node* newNode;
3     newNode=(struct node*) malloc(sizeof(struct node));
4     newNode->data = newData;
5     newNode->next = *headRef;
6     *headRef = newNode;
7 }
```

H Indata är referensen till listan, `headRef` samt ett tal `newData`.

3 Allokerar ett nytt element på heapen

4 Tilldelar posten data

5 Länkar in det nya lementet först i listan

6 Ger listan en ny startadress

```
1  int GetNth(struct node* head, int index) {
2      struct node* current = head;
3      int count=0;
4      while (current!=NULL) {
5          if (count==index) return(current->data);
6          count++;
7          current=current->next;
8      }
9      assert(0);
10 }
```

H Returnerar värdet hos elementet som befinner sig på plats `index` i listan. Första platsen har index `0`.

3 Räknaren som håller reda på var vi befinner oss.

4 Så länge listan inte är slut kan vi fortsätta stega framåt

5 När vi nått fram till aktuellt element returnerar vi värdet

9 Når vi hit, är listan för kort och programmet stannar med felmeddelande

```
1  void DeleteList(struct node** headRef) {
2      struct node* current = *headRef;
3      struct node* next;
4      while (current != NULL) {
5          next = current->next;
6          free(current);
7          current = next;
8      }
9      *headRef = NULL;
10 }
```

H Tar bort hela listan. Detta måste göras i rätt ordning. `free` heter funktionen som återlämnar minnet till heapen. Om man inte sköter detta korrekt kommer heapen att innehålla upptaget, ej använt minne – minnesläcka.

1 Indata är adressen till den plats där adressen till listans start ligger.

9 När alla element är borttagna ska ska förstås adressen till listan vara NULL.

```
1  void InsertNth(struct node** headRef, int index, int data) {
2      if (index == 0)
3          Push(headRef, data);
4      else {
5          struct node* current = *headRef;
6          int i;
7          for (i=0; i<index-1; i++) {
8              assert(current != NULL);
9              current = current->next;
10         }
11         assert(current != NULL);
12         Push(&(current->next), data);
13     }
14 }
```

H Skapar ett element på plats `index` med värdet `data`.

2 Om elementet ska stå först i listan betraktar vi det som ett specialfall, eftersom `headRef` då ska uppdateras.

5 `current` pekar ut listans start

7 `for`-loopen söker upp den plats där elementet ska in.

8 Om `index` är större än listans längd, kan inte uppgiften utföras och programmet stannar med felmeddelande

11 `current` får inte vara `NULL` här heller.

12 ett avancerat sätt att använda `Push`

```
1  void SortedInsert(struct node** headRef, struct node* newNode) {
2      if (*headRef == NULL || (*headRef)->data >= newNode->data) {
3          newNode->next = *headRef;
4          *headRef = newNode;
5      } else {
6          struct node* current = *headRef;
7          while (current->next!=NULL && current->next->data<newNode->data) {
8              current = current->next;
9          }
10         newNode->next = current->next;
11         current->next = newNode;
12     }
13 }
```

H Instickssortering. Funktionen söker upp den plats där givet element ska placeras in för att listan ska fortsätta att vara sorterad i stigande ordning

1 Först behandlas de två specialfallen då listan är tom och då elementet hamnar först i listan. Egentligen samma sak.

7-9 Letar upp den plats där elementet ska in. Antingen tar listan slut eller så är `newNode->data` större än `data` i nästa element.

10-11 Elementet placeras in på rätt plats.

N Nedan ser vi ett exempel på hur funktionen kan användas. Resultatet av utskriften blir `5,10,12`.

```
int main(void) {
    struct node *head=NULL,*ny;
    Push(&head,10);
    ny=(struct node*) malloc(sizeof(struct node));
    ny->data=5;
    SortedInsert(&head,ny);
    ny=(struct node*) malloc(sizeof(struct node));
    ny->data=12;
    SortedInsert(&head,ny);
    Print(head);
}
```

```
void SortedInsert2(struct node** headRef, struct node* newNode) {
    struct node dummy;
    struct node* current = &dummy;
    dummy.next = *headRef;
    while (current->next!=NULL && current->next->data<newNode->data) {
        current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
    *headRef = dummy.next;
}
```

H En alternativ funktion för instickssortering.

```
void SortedInsert3(struct node** headRef, struct node* newNode) {
    struct node** currentRef = headRef;
    while (*currentRef!=NULL && (*currentRef)->data<newNode->data) {
        currentRef = &((*currentRef)->next);
    }
    newNode->next = *currentRef;
    *currentRef = newNode;
}
```

H Ytterligare ett annat sätt att utföra instickssortering.

```
int Pop(struct node** headRef) {
    struct node* head;
    int result;
    head=*headRef;
    assert(head!=NULL);
    result=head->data;
    *headRef = head->next;
    free(head);
    return result;
}
```

H `Pop` returnerar värdet hos det första elementet i listan och plockar bort det.

```
1  void InsertSort(struct node** headRef) {
2      struct node* result = NULL;
3      struct node* current = *headRef;
4      struct node* next;
5      while (current!=NULL) {
6          next = current->next;
7          SortedInsert(&result, current);
8          current = next;
9      }
10     *headRef = result;
11 }
```

H Sorterar en länkad lista med hjälp av instickssortering.

```
1  void Append(struct node** aRef, struct node** bRef) {
2      struct node* current;
3      if (*aRef == NULL) {
4          *aRef = *bRef;
5      } else {
6          current = *aRef;
7          while (current->next != NULL) {
8              current = current->next;
9          }
10         current->next = *bRef;
11     }
12     *bRef=NULL;
13 }
```

H Lägger samman två listor. 'appendar'.

4 Specialfall då första listan, `aRef` är tom.

7-9 Sök upp slutet av den första listan, `aRef`.

10 Skarva.

12 Lista `bRef` finns inte längre.

```
1  void FrontBackSplit(struct node* source,
2                      struct node** frontRef, struct node** backRef) {
3      int len = Length(source);
4      int i;
5      struct node* current = source;
6      if (len < 2) {
7          *frontRef = source;
8          *backRef = NULL;
9      } else {
10         int hopCount = (len-1)/2;
11         for (i = 0; i<hopCount; i++) {
12             current = current->next;
13         }
14         *frontRef = source;
15         *backRef = current->next;
16         current->next = NULL;
17     }
18 }
```

H Delar en lista i två lika långa. Om udda antal ska den första listan, `frontRef` ha ett element mer.

3 Vi använder `Length` för att ta reda på antalet element i den ursprungliga listan.

6 Specialfall då listan är tom eller endast har ett element.

11-13 Söker upp mitten.

12-14 Dela listan och sätt in `NULL` i slutet på första listan.

```
1  void FrontBackSplit2(struct node* source,
2                       struct node** frontRef, struct node** backRef) {
3      struct node* fast;
4      struct node* slow;
5      if (source==NULL || source->next==NULL) {
6          *frontRef = source;
7          *backRef = NULL;
8      } else {
9          slow = source;
10         fast = source->next;
11         while (fast != NULL) {
12             fast = fast->next;
13             if (fast != NULL) {
14                 slow = slow->next;
15                 fast = fast->next;
16             }
17         }
18         *frontRef = source;
19         *backRef = slow->next;
20         slow->next = NULL;
21     }
22 }
```

H En alternativ lösning av `FrontBackSplit`

```
1  void RemoveDuplicates(struct node* head) {
2      struct node* current = head;
3      if (current == NULL) return;
4      while (current->next!=NULL) {
5          if (current->data == current->next->data) {
6              struct node* nextNext = current->next->next;
7              free(current->next);
8              current->next = nextNext;
9          } else {
10             current = current->next;
11         }
12     }
13 }
```

H Tar bort dubbletter i en i övrigt sorterad lista

```
void MoveNode(struct node** destRef, struct node** sourceRef) {
    struct node* newNode = *sourceRef;
    assert(newNode != NULL);
    *sourceRef = newNode->next;
    newNode->next = *destRef;
    *destRef = newNode;
}
```

H Flyttar första elementet i `sourceRef` som första element i `destRef`, istället för att använda `Pop` och `Push`

```
void AlternatingSplit(struct node* source,
                      struct node** aRef, struct node** bRef) {
    struct node* a = NULL;
    struct node* b = NULL;
    struct node* current = source;
    while (current != NULL) {
        MoveNode(&a, &current);
        if (current != NULL) {
            MoveNode(&b, &current);
        }
    }
    *aRef = a;
    *bRef = b;
}
```

H Delar upp en lista i två, där alla med jämnt index hamnar i den `aRef` och de med udda i `bRef`.

```
void AlternatingSplit2(struct node* source,
                       struct node** aRef, struct node** bRef) {
    struct node aDummy;
    struct node* aTail = &aDummy;
    struct node bDummy;
    struct node* bTail = &bDummy;
    struct node* current = source;
    aDummy.next = NULL;
    bDummy.next = NULL;
    while (current != NULL) {
        MoveNode(&(aTail->next), &current);
        aTail = aTail->next;
        if (current != NULL) {
            MoveNode(&(bTail->next), &current);
            bTail = bTail->next;
        }
    }
    *aRef = aDummy.next;
    *bRef = bDummy.next;
}
```

H En alternativ lösning

```
struct node* ShuffleMerge1(struct node* a, struct node* b) {
    struct node dummy;
    struct node* tail = &dummy;
    dummy.next = NULL;
    while (1) {
        if (a==NULL) {
            tail->next = b;
            break;
        } else if (b==NULL) {
            tail->next = a;
            break;
        } else {
            tail->next = a;
            tail = a;
            a = a->next;
            tail->next = b;
            tail = b;
            b = b->next;
        }
    }
    return(dummy.next);
}
```

H Tar emot två listor förenas till en genom 'riffelblandning'Í. Till exempel $[a, a, a]$ och $[b, b, b]$ övergår i listan $[a, b, a, b, a, b]$.

```
struct node* ShuffleMerge2(struct node* a, struct node* b) {
    struct node dummy;
    struct node* tail = &dummy;
    dummy.next = NULL;
    while (1) {
        if (a==NULL) {
            tail->next = b;
            break;
        } else if (b==NULL) {
            tail->next = a;
            break;
        } else {
            MoveNode(&(tail->next), &a);
            tail = tail->next;
            MoveNode(&(tail->next), &b);
            tail = tail->next;
        }
    }
    return(dummy.next);
}
```

H Alternativ lösning till 'riffelblandning'

```
struct node* ShuffleMerge3(struct node* a, struct node* b) {
    struct node* result = NULL;
    struct node** lastPtrRef = &result;
    while (1) {
        if (a==NULL) {
            *lastPtrRef = b;
            break;
        } else if (b==NULL) {
            *lastPtrRef = a;
            break;
        } else {
            MoveNode(lastPtrRef, &a);
            lastPtrRef = &((*lastPtrRef)->next);
            MoveNode(lastPtrRef, &b);
            lastPtrRef = &((*lastPtrRef)->next);
        }
    }
    return(result);
}
```

H Ytterligare en 'riffelblandning'

```
struct node* ShuffleMerge4(struct node* a, struct node* b) {
    struct node* result;
    struct node* recur;
    if (a==NULL) return(b);
    else if (b==NULL) return(a);
    else {
        recur = ShuffleMerge4(a->next, b->next);
        result = a;
        a->next = b;
        b->next = recur;
        return(result);
    }
}
```

H En rekursiv lösning på 'riffelblandning'

```
struct node* SortedMerge(struct node* a, struct node* b) {
    struct node dummy;
    struct node* tail = &dummy;
    dummy.next = NULL;
    while (1) {
        if (a == NULL) {
            tail->next = b;
            break;
        } else if (b == NULL) {
            tail->next = a;
            break;
        }
        if (a->data <= b->data) {
            MoveNode(&(tail->next), &a);
        } else {
            MoveNode(&(tail->next), &b);
        }
        tail = tail->next;
    }
    return(dummy.next);
}
```

H Samsortering av två sorterade listor

```
struct node* SortedMerge2(struct node* a, struct node* b) {
    struct node* result = NULL;
    struct node** lastPtrRef = &result;
    while (1) {
        if (a==NULL) {
            *lastPtrRef = b;
            break;
        } else if (b==NULL) {
            *lastPtrRef = a;
            break;
        }
        if (a->data <= b->data) {
            MoveNode(lastPtrRef, &a);
        } else {
            MoveNode(lastPtrRef, &b);
        }
        lastPtrRef = &((*lastPtrRef)->next);
    }
    return(result);
}
```

H En annan lösning av samsortering.

```
struct node* SortedMerge3(struct node* a, struct node* b) {
    struct node* result = NULL;
    if (a==NULL) return(b);
    else if (b==NULL) return(a);
    if (a->data <= b->data) {
        result = a;
        result->next = SortedMerge3(a->next, b);
    } else {
        result = b;
        result->next = SortedMerge3(a, b->next);
    }
    return(result);
}
```

H En rekursiv lösning av samsortering.

```
void MergeSort(struct node** headRef) {
    struct node* head = *headRef;
    struct node* a;
    struct node* b;
    if ((head == NULL) || (head->next == NULL)) {
        return;
    }
    FrontBackSplit(head, &a, &b);
    MergeSort(&a);
    MergeSort(&b);
    *headRef = SortedMerge(a, b);
}
```

H MergeSort på listor. Vi återkommer till metoden längre fram i kursen, i samband med *Söndra och Härska*.

N Bygg upp en lista med 100000 slumptal och sortera den, först med `InsertSort` och sedan med `MergeSort`. Jämför exekveringstiderna!

```
    struct node dummy;
    struct node* tail = &dummy;
    dummy.next = NULL;
    while (a!=NULL && b!=NULL) {
        if (a->data == b->data) {
            Push((&tail->next), a->data);
            tail = tail->next;
            a = a->next;
            b = b->next;
        } else if (a->data < b->data) {
            a = a->next;
        } else {
            b = b->next;
        }
    }
    return(dummy.next);
}
```

H Ta fram snittet av två sorterade listor, det vill säga de elementvärden som finns i båda listorna. Resultatet hamnar i en ny lista.

```c
struct node* SortedIntersect2(struct node* a, struct node* b) {
    struct node* result = NULL;
    struct node** lastPtrRef = &result;
    while (a!=NULL && b!=NULL) {
        if (a->data == b->data) {
            Push(lastPtrRef, a->data);
            lastPtrRef = &((*lastPtrRef)->next);
            a=a->next;
            b=b->next;
        } else if (a->data < b->data) {
            a=a->next;
        } else {
            b=b->next;
        }
    }
    return(result);
}
```

H En annan lösning på samma problem.

```c
static void Reverse(struct node** headRef) {
    struct node* result = NULL;
    struct node* current = *headRef;
    struct node* next;
    while (current != NULL) {
        next = current->next;
        current->next = result;
        result = current;
        current = next;
    }
    *headRef = result;
}
```

H Vänder på en lista

```c
static void Reverse2(struct node** headRef) {
    struct node* result = NULL;
    struct node* current = *headRef;
    while (current != NULL) {
        MoveNode(&result, &current);
    }
    *headRef = result;
}
```

H En annan lösning som använder sig av `MoveNode`

```
void Reverse3(struct node** headRef) {
    if (*headRef != NULL) {
        struct node* middle = *headRef;
        struct node* front = middle->next;
        struct node* back = NULL;
        while (1) {
            middle->next = back;
            if (front == NULL) break;
            back = middle;
            middle = front;
            front = front->next;
        }
        *headRef = middle;
    }
}
```

H En tredje lösning, som använder sig av tre pekare.

```
void RecursiveReverse(struct node** headRef) {
    struct node* first;
    struct node* rest;
    if (*headRef == NULL) return;
    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);
    first->next->next = first;
    first->next = NULL;
    *headRef = rest;
}
```

H En rekursiv lösning

```
void Print(struct node* headRef){
    while(headRef!=NULL){
        printf("%d ",headRef->data);
        headRef=headRef->next;
    }
    printf("\n");
}
```

H Skriver ut en lista.