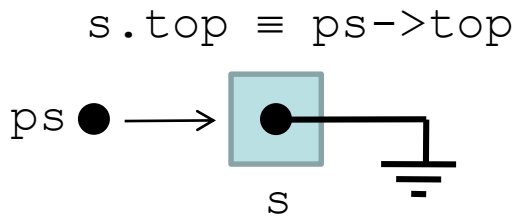
Data Structure

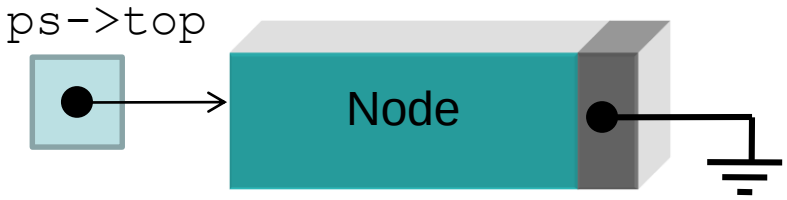
Lecture 3

Dr. Ahmed Fathalla

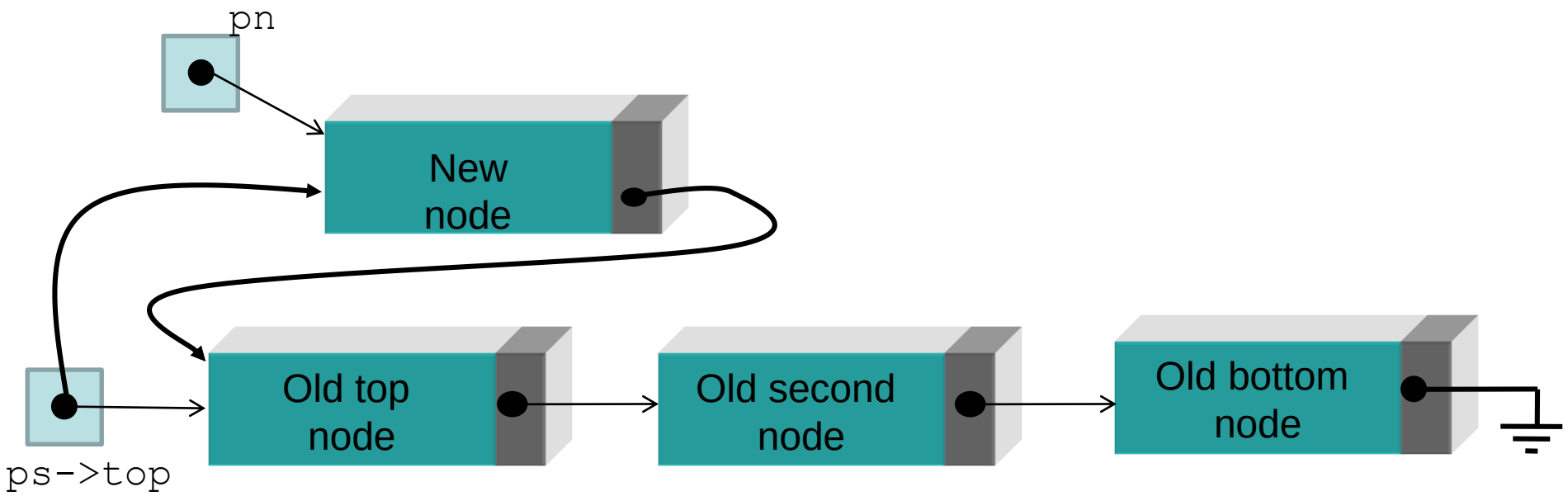
Linked-based implementation
(to overcome fixed size limitations):



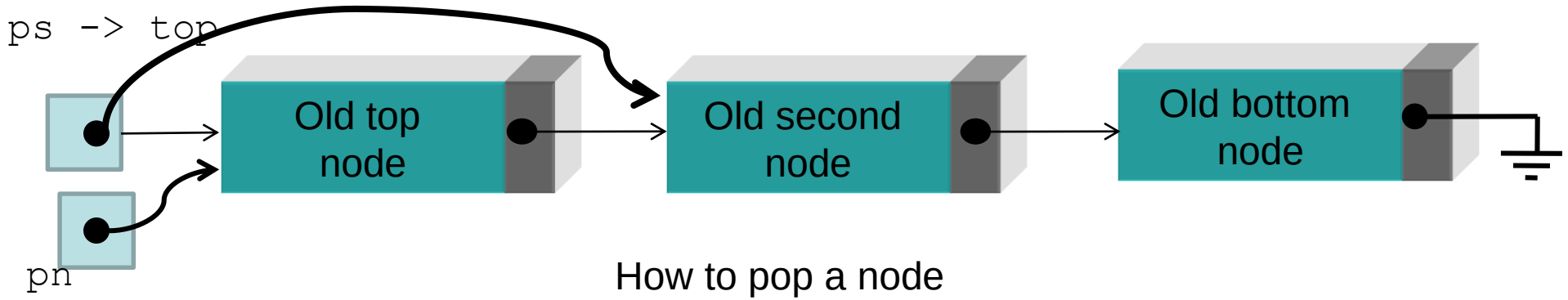
Empty stack



Stack of size 1

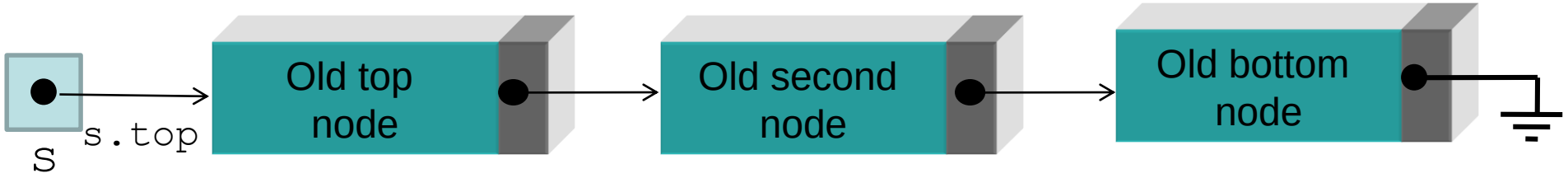


How to push a node



How to pop a node

Type Definition




```
struct stacknode{
    StackEntry entry;
    struct stacknode *next;
};

struct stack{
    StackNode *top;
};
```

Why not:

```
Typedef StackNode *Stack;
```

1. To make *logical distinction* between the stack itself and its top, which points to a node.

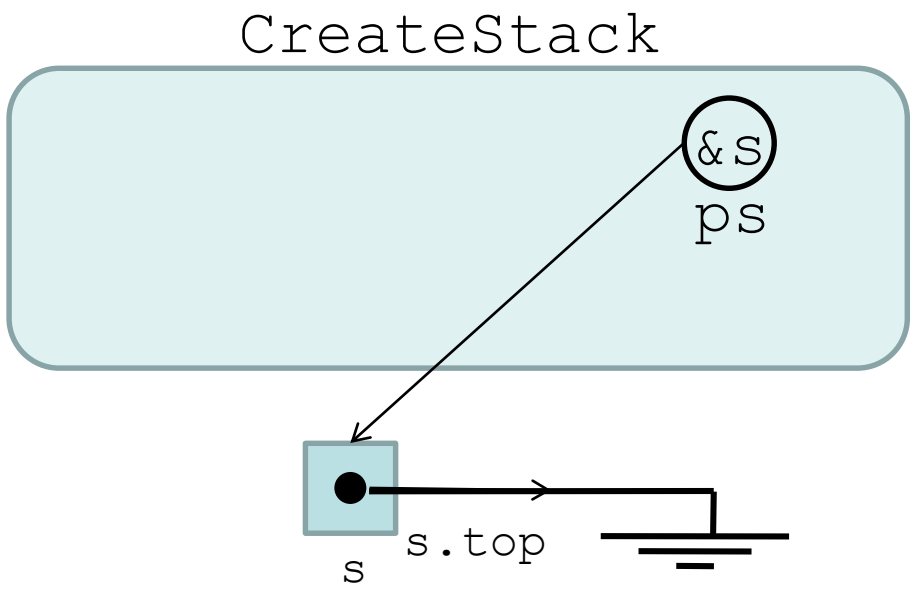
2. To be consistent with the definitions of other DS.

3. For upgradability (adding more functions) that may need other pieces of information to be saved than `top`. (we will see).

Implementation level (what really happens)

```
void CreateStack(Stack *ps){
   ps->top=NULL;
}
```

`NULL` is defined in `<stdlib.h>`



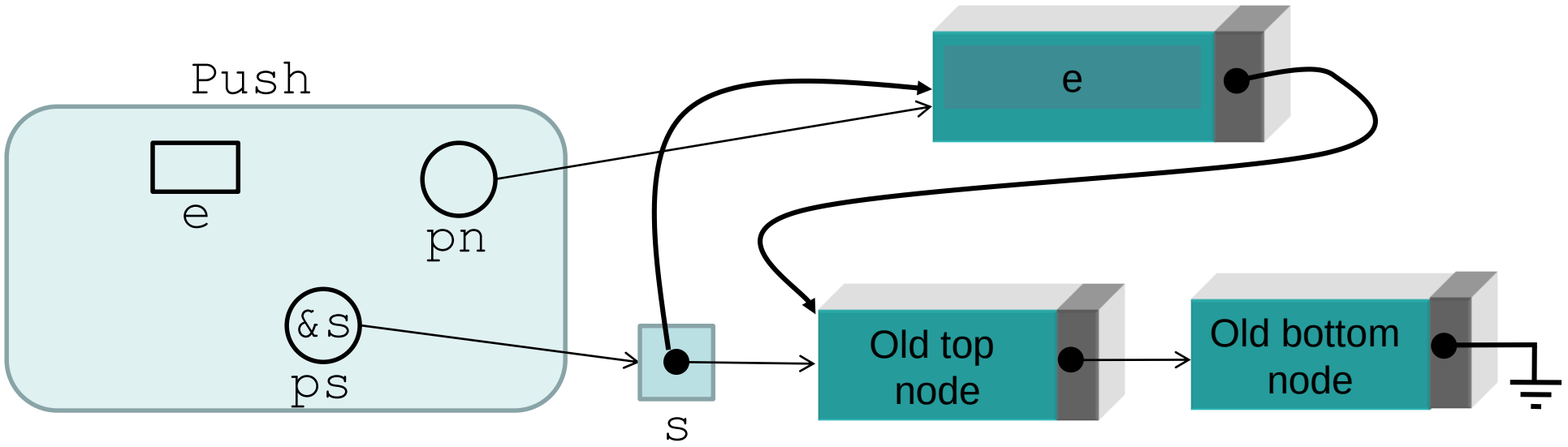
User Level (interface)

```
void main(){

Stack s;

CreateStack(&s);



}
```

```
/* Pre: The stack exists and is initialized.
   Post: The argument item has been stored at the top
   of the stack */

void Push(StackEntry e, Stack *ps){
  StackNode *pn = new StackNode;
  pn->entry=e;
  pn->next=ps->top;
  ps->top=pn;
}
```

User Call:

```
StackEntry e;
Stack s;
     :
CreateStack(&s);
     :
Push(e, &s);
```



Always take care of special cases

What if the Stack is Empty?

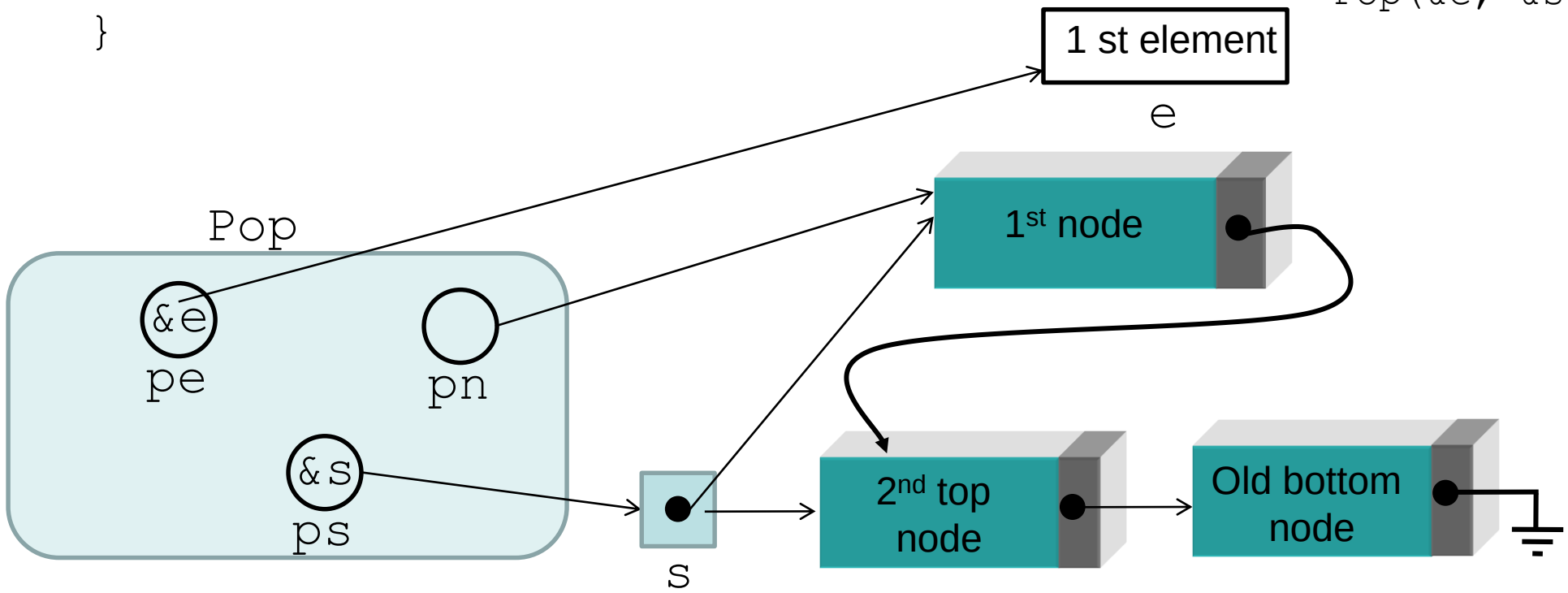
```
/* Pre: The stack exists and is initialized
   Post: The argument item has been stored at the top
   of the stack */

void Push(StackEntry e, Stack *ps){
  StackNode *pn = new StackNode;
  pn->entry=e;
  pn->next=ps->top;
  ps->top=pn;
}
```

```
/*Pre: The stack exists and it is not empty.
  Post: The item at the top of the stack has been
  removed and returned in *pe  */

void Pop(StackEntry *pe, Stack *ps){
  StackNode *pn;
  *pe=ps->top->entry;
  pn=ps->top;
  ps->top=ps->top->next;
  delete pn;
}
```

This is just the code for `StackTop`.

User Call:

```
StackEntry e;
Stack s;
     :
CreateStack(&s);
     :
Pop(&e, &s);
```



Always take care of special cases

What if the Stack is Empty?

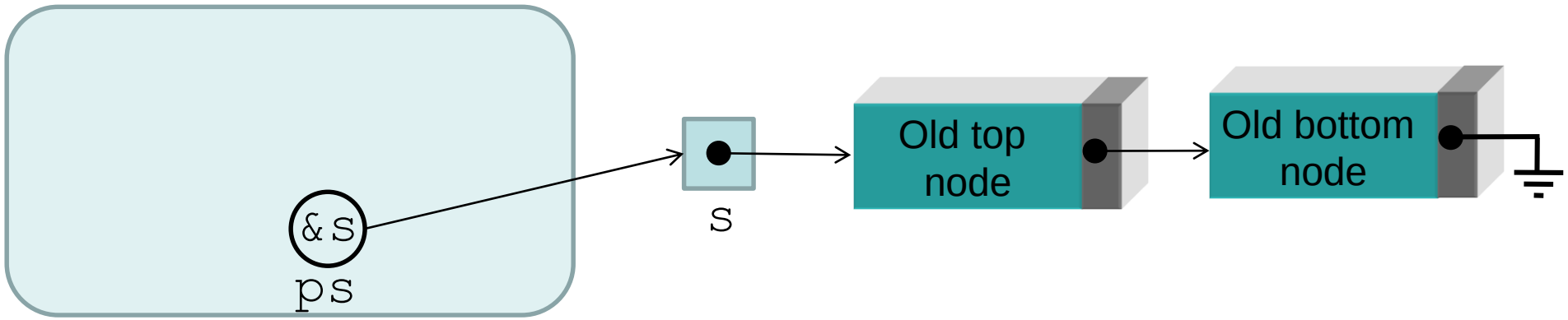
```
/*Pre: The stack exists and it is not empty.
  Post: The item at the top of the stack has been
  removed and returned in *pe  */

void Pop(StackEntry *pe, Stack *ps){
      StackNode *pn;
      *pe=ps->top->entry;
      pn=ps->top;
      ps->top=ps->top->next;
      delete pn;
}
```

User Call:

```
StackEntry e;
Stack s;
      :
CreateStack(&s);
     :


}
```

```
/* Pre: The stack exists
   Post: returns the status, 1 or 0*/

int StackEmpty(Stack *ps){
      return ps->top==NULL;
}

int StackFull(Stack *ps){
      return 0;
}
```

For any function that does not change the stack there is no problem in passing the stack itself `s` rather than a pointer to it `ps`. This will not copy the elements as opposed to the array-based implementation. **However, of course,** we do not do that to keep the code at the **user level** unchanged.



```
/* Pre: The stack exists
Post: All the elements freed
*/
void ClearStack(Stack *ps){
  StackNode *pn=ps->top;
  StackNode *qn=ps->top;
  while(pn){
    pn=pn->next;
    delete qn;
    qn=pn;
  }
  ps->top=NULL;
}
```

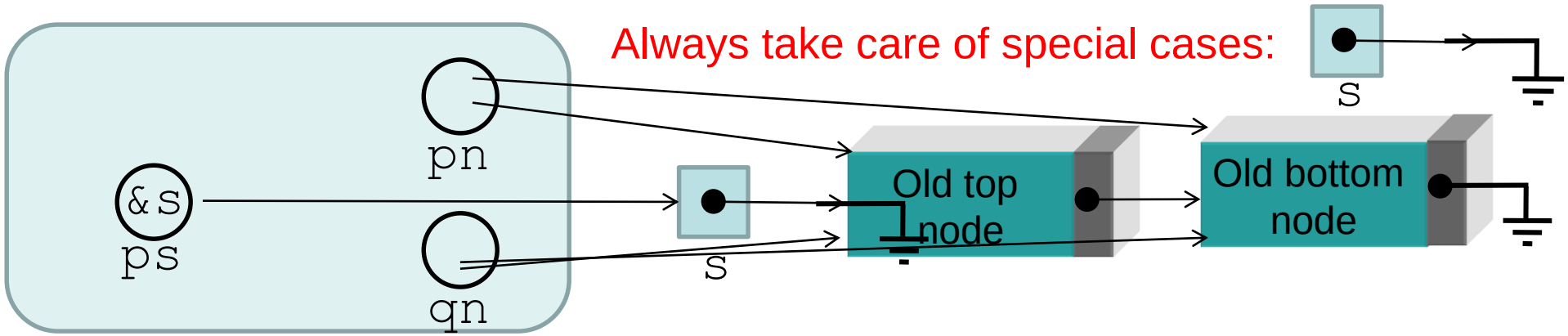
The same as: `pn!=NULL`

You can replace `qn` by `ps->top`

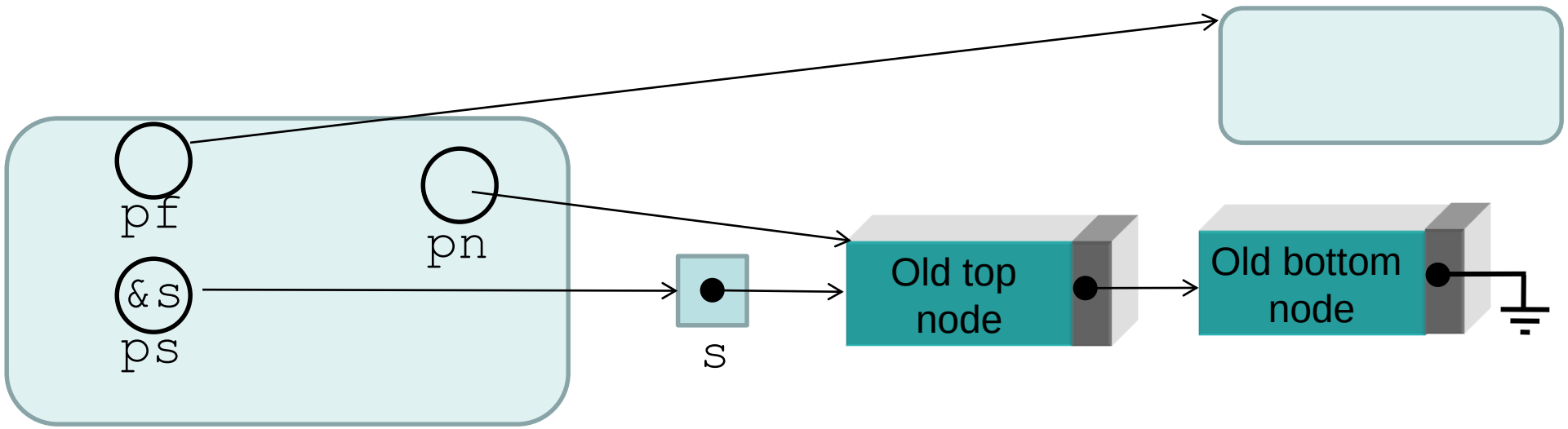
```
void ClearStack(Stack *ps){
  StackNode *pn=ps->top;

  while(pn){
    pn=pn->next;
    delete ps->top;
    ps->top=pn;
  }

}
```

The wrong code is:

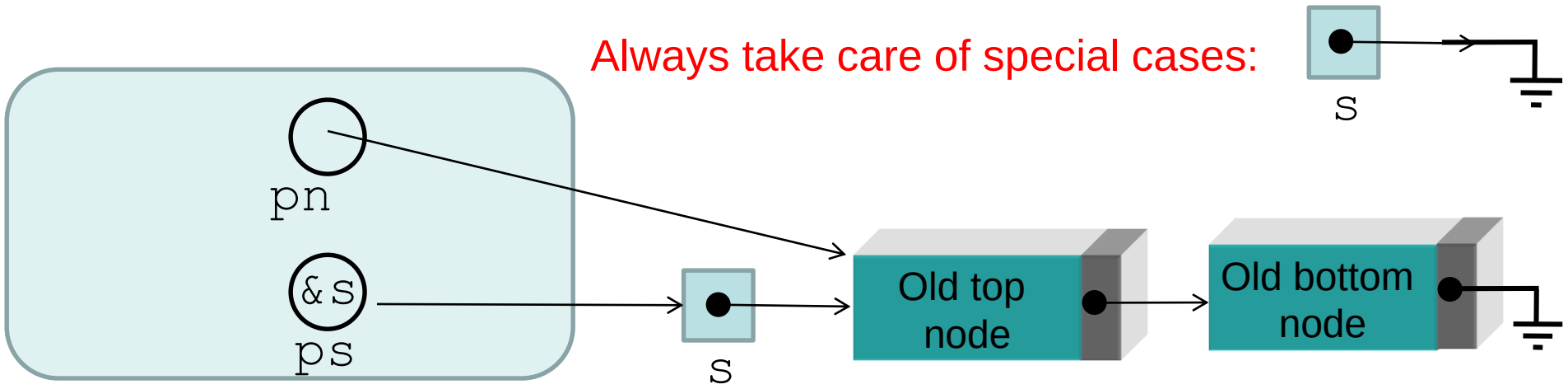
```
ps->top=NULL;
```



```
/* Pre: The stack exists
   Post: Function is passed to process every element*/

void TraverseStack(Stack *ps){
  StackNode *pn=ps->top;
  while(pn){
    cout<<pn->entry<<" ";
    pn=pn->next;
  }
}
```



```
/* Pre: The stack exists
   Post: returns the number of elements*/

int   StackSize(Stack *ps){
  int x;
  StackNode *pn=ps->top;
  for(x=0; pn; pn=pn->next)
    x++;
  return x;
}
```



We add extra field, called, `size` in `struct stack`. Then, we need to add just one statement to: `CreateStack, Pop, Push, ClearStack`.

```
typedef struct stack{
  StackNode *top;
  int  size;
}Stack;

void CreateStack(Stack *ps){
  ps->top=NULL;
  ps->size=0;
}

void Push(StackEntry e, Stack *ps){
  StackNode   *pn = new StackNode;
  pn->entry=e;
  pn->next=ps->top;
  ps->top=pn;
  ps->size++;
}
```

```
void Pop(StackEntry *pe, Stack *ps){
  StackNode *pn;
  *pe=ps->top->entry;
  pn=ps->top;
  ps->top=ps->top->next;
  delete pn;
  ps->size--;
}

void ClearStack(Stack *ps){
  StackNode *pn=ps->top;
  while(pn){
    pn=pn->next;
    delete ps->top;
    ps->top=pn;
  }
  ps->size=0;
}
```

Then the function StackSize is simply:

```
int StackSize(Stack *ps){
  return ps->size;
}
```