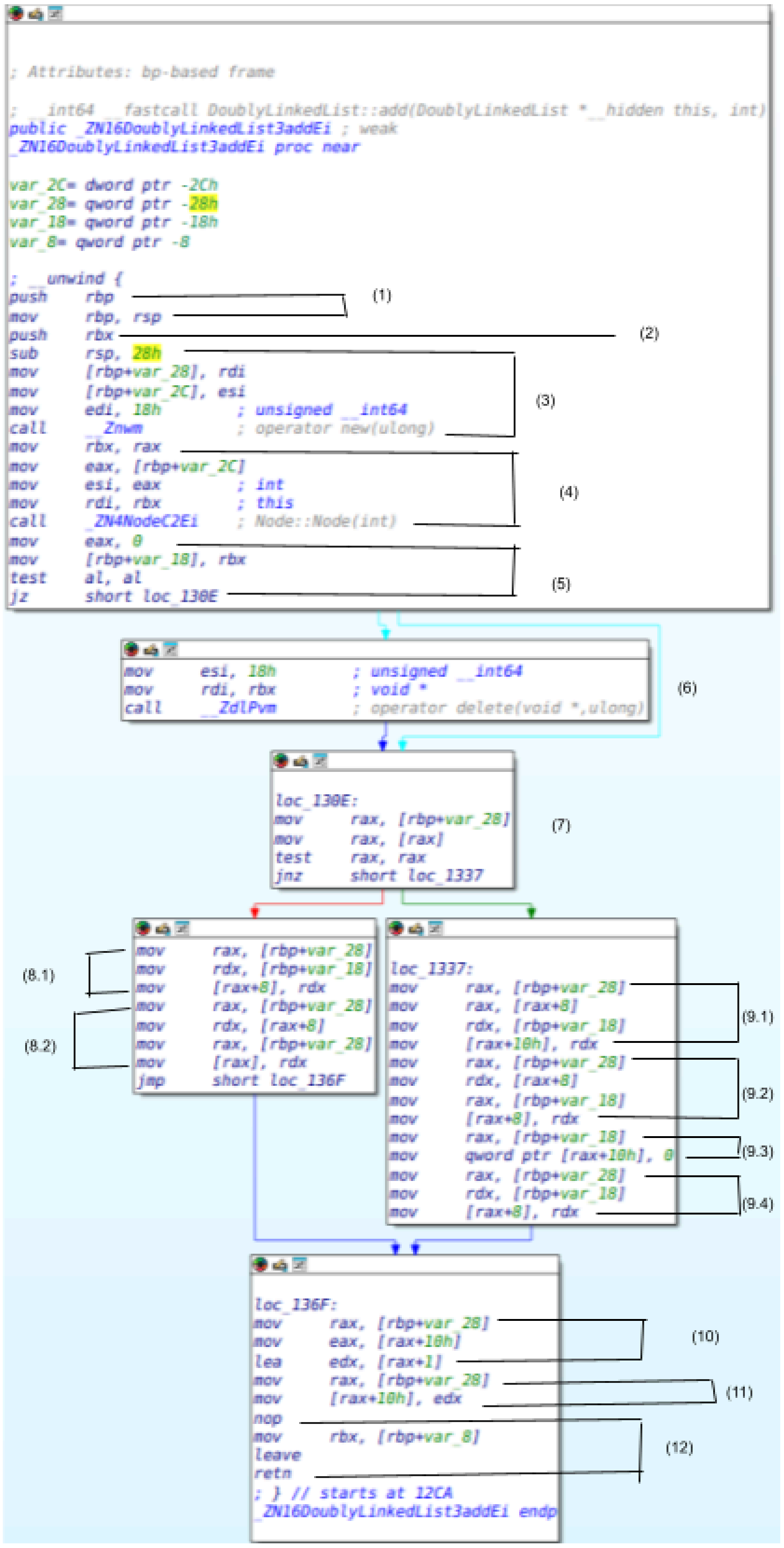

```
; Attributes: bp-based frame

; __int64 __fastcall DoublyLinkedList::add(DoublyLinkedList *__hidden this, int)
public _ZN16DoublyLinkedList3addEi ; weak
_ZN16DoublyLinkedList3addEi proc near

var_2C= dword ptr -2Ch
var_28= qword ptr -28h
var_18= qword ptr -18h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 28h
mov     [rbp+var_28], rdi
mov     [rbp+var_2C], esi
mov     edi, 18h        ; unsigned __int64
call    __Znwm          ; operator new(ulong)
mov     rbx, rax
mov     eax, [rbp+var_2C]
mov     esi, eax        ; int
mov     rdi, rbx        ; this
call    _ZN4NodeC2Ei    ; Node::Node(int)
mov     eax, 0
mov     [rbp+var_18], rbx
test    al, al
jz      short loc_130E
```

```
mov     esi, 18h        ; unsigned __int64
mov     rdi, rbx        ; void *
call    __ZdlPvm        ; operator delete(void *,ulong)
```

```
loc_130E:
mov     rax, [rbp+var_28]
mov     rax, [rax]
test    rax, rax
jnz     short loc_1337
```

```
mov     rax, [rbp+var_28]
mov     rdx, [rbp+var_18]
mov     [rax+8], rdx
mov     rax, [rbp+var_28]
mov     rdx, [rax+8]
mov     rax, [rbp+var_28]
mov     [rax], rdx
jmp     short loc_136F
```

```
loc_1337:
mov     rax, [rbp+var_28]
mov     rax, [rax+8]
mov     rdx, [rbp+var_18]
mov     [rax+10h], rdx
mov     rax, [rbp+var_28]
mov     rdx, [rax+8]
mov     rax, [rbp+var_18]
mov     [rax+8], rdx
mov     rax, [rbp+var_18]
mov     qword ptr [rax+10h], 0
mov     rax, [rbp+var_28]
mov     rdx, [rbp+var_18]
mov     [rax+8], rdx
```

```
loc_136F:
mov     rax, [rbp+var_28]
mov     eax, [rax+10h]
lea     edx, [rax+1]
mov     rax, [rbp+var_28]
mov     [rax+10h], edx
nop
mov     rbx, [rbp+var_8]
leave
retn
; } // starts at 12CA
_ZN16DoublyLinkedList3addEi endp
```

1. Function Preamble
2. I don't fully understand why this operation is done, my assumption is that it's pushed onto the stack for easy memory access for creating the Node object.
3. Memory allocation step. We attempt to grab memory to allocate parts of the node to before calling new (ptrs for prev and next are in rdi while our called val is in esi). We allocate 18 hex bytes worth of space for our new node.
4. Grabs the memory address allocated from new and sets eax to the passed in value for our data. It then calls the Node constructor.
5. Checks to see if the constructor succeeds.
6. This is a fallback case in case our constructor fails or aborts. It will call the delete operator. Since I didn't create a case for this in my source code, it goes into the add operations anyways. This could be an improvement to my existing code.
7. This loads in the head node and checks if it exists before going into the branching if else case through jnz.
8. This is where head doesn't exist
    - 8.1
        1. This puts the location of head into a register and stores the head node at the same address as the newnode.
    - 8.2
        1. This loads a new segment of memory (where tail will be stored) and stores the newnode addr into where the tail will be.
9. This is the else case. (default)
    - 9.1
        1. Loads in the tail node and then loads in the newnode to be pointed by tail's next
    - 9.2
        1. Loads in tail and then sets newnode's prev value to point to it.
    - 9.3
        1. Sets newnode's next value to point to null
    - 9.4
        1. Equates the tail to point to the new node added to update the state of the list.
10. Loads in the list from its location in memory to a register and increments edx by 1 from the list's length.
11. Loads in the list again and sets its the new len to where the list's len is located in memory.
12. Function Epilogue. RBX is set to a new chunk of memory to be utilized when add is called for adding a new node to the list.